Trademarks

RTI, Real-Time Innovations, Connext, NDDS, the RTI logo, 1RTI and the phrase, “Your Systems. Working as one,” are registered trademarks, trademarks or service marks of Real-Time Innovations, Inc. All other trademarks belong to their respective owners.

Copy and Use Restrictions

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form (including electronic, mechanical, photocopy, and facsimile) without the prior written permission of Real-Time Innovations, Inc. The software described in this document is furnished under and subject to the RTI software license agreement. The software may be used or copied only under the terms of the license agreement.

This is an independent publication and is neither affiliated with, nor authorized, sponsored, or approved by, Microsoft Corporation.

The security features of this product include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com).

Notice

Any deprecations noted in this document serve as notice under the Real-Time Innovations, Inc. Maintenance Policy #4220.

Technical Support

Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089
Phone: (408) 990-7444
Email: support@rti.com
Website: https://support.rti.com/
Available Documentation

To get you up and running as quickly as possible, the RTI® Connext® DDS documentation is divided into several parts.

- **RTI Connext DDS Installation Guide** — This document describes how to install Connext DDS, as well as License Management.

- **RTI Connext DDS Getting Started Guide** — This document describes the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application. Addendums cover:
  - RTI Connext DDS Core Libraries Getting Started Guide Addendum for Android Systems
  - RTI Connext DDS Core Libraries Getting Started Guide Addendum for Embedded Systems
  - RTI Connext DDS Core Libraries Getting Started Guide Addendum for INtime Systems

- **RTI Connext DDS Core Libraries What's New in 6.1.1** — This document describes changes and enhancements in the most recent major release of Connext. Those upgrading from a previous version should read this document first.

- **RTI Connext DDS Core Libraries Release Notes** — This document describes system requirements, what's fixed, and known issues.

- **RTI Connext DDS Core Libraries Platform Notes** — This document provides platform-specific information, including specific information required to build your applications using Connext DDS, such as compiler flags and libraries.

- **RTI Connext DDS Core Libraries Database Setup** — In most cases, Connext DDS does not require a database. However, if you want to use some of the durability features of Connext DDS, you will need a relational database.
• *Migration Guide* on the RTI Community Portal ([https://community.rti.com/documentation](https://community.rti.com/documentation))—This document describes how to migrate to the current release from a previous *Connext DDS* release, including what compatibility issues you may need to account for during your upgrade. This guide is updated as needed.

• *RTI Connext DDS Core Libraries User's Manual* — This document describes the features of the product and how to use them. It is organized around the structure of the *Connext DDS* APIs and certain common high-level tasks.

• *RTI Connext DDS Core Libraries Extensible Types Guide* — This document describes how to use Extensible Types, which allow you to define data types in a more flexible way, and to evolve data types over time without giving up portability, interoperability, or the expressiveness of the DDS type system.

• *API Reference HTML Documentation* ([README.html](https://community.rti.com/documentation)) — This extensively cross-referenced documentation, available for all supported programming languages, is your in-depth reference to every operation and configuration parameter in the middleware. Even experienced *Connext DDS* developers will often consult this information.

• **The Programming How To's provide a good place to begin learning the APIs.** These are hyper-linked code snippets to the full API documentation. From the [README.html](https://community.rti.com/documentation) file, select one of the supported programming languages, then scroll down to the Programming How To’s. Start by reviewing the Publication Example and Subscription Example, which provide step-by-step examples of how to send and receive data with *Connext DDS*.

Many readers will also want to look at additional documentation available online. In particular, RTI recommends the following:

• Use the [RTI Customer Portal](http://support.rti.com) to download RTI software and contact RTI Support. The RTI Customer Portal requires a username and password. You will receive this in the email confirming your purchase. If you do not have this email, please contact [license@rti.com](mailto:license@rti.com). Resetting your login password can be done directly at the RTI Customer Portal.

• The RTI Community Portal ([https://community.rti.com](https://community.rti.com)) provides a wealth of knowledge to help you use *Connext DDS*, including:
  • Documentation, at [https://community.rti.com/documentation](https://community.rti.com/documentation)
  • Best Practices
  • Example code for specific features, as well as more complete use-case examples,
  • Solutions to common questions,
  • A glossary,
  • Downloads of experimental software,
  • And more.
Whitepapers and other articles are available from http://www.rti.com/resources.
# Contents

## About this Document

- Paths Mentioned in Documentation ................................................................. 1
- Programming Language Conventions ............................................................... 2
  - Traditional vs. Modern C++ ................................................................. 2
- Extensions to the DDS Standard ...................................................................... 3
- Environment Variables .................................................................................... 3
- Additional Resources ....................................................................................... 4

## Part 1: Welcome to RTI Connext DDS ................................................................. 5

### Chapter 1 Overview

- 1.1 What is Connext DDS? .............................................................................. 6
- 1.2 Network Communications Models ............................................................ 7
- 1.3 What is Middleware? .................................................................................. 10
- 1.4 Features of Connext DDS ........................................................................ 11

## Chapter 2 Data-Centric Publish-Subscribe Communications

- 2.1 What is DCPS? ......................................................................................... 14
  - 2.1.1 DCPS for Real-Time Requirements ....................................................... 15
- 2.2 DDS Data Types, Topics, Keys, Instances, and Samples .............................. 16
- 2.3 Data Topics — What is the Data Called? .................................................... 17
- 2.4 DDS Samples, Instances, and Keys ............................................................. 18
- 2.5 DataWriters/Publishers and DataReaders/Subscribers ................................. 20
- 2.6 DDS Domains and DomainParticipants ..................................................... 22
- 2.7 Quality of Service (QoS) ........................................................................... 23
  - 2.7.1 Controlling Behavior with Quality of Service (QoS) Policies .................... 23
- 2.8 Application Discovery .............................................................................. 24

## Part 2: Core Concepts ....................................................................................... 26

### Chapter 3 Data Types and DDS Data Samples
3.1 Introduction to the Type System ................................................................. 29
  3.1.1 Sequences ......................................................................................... 30
  3.1.2 Strings and Wide Strings ................................................................. 32
  3.1.3 Introduction to TypeCode ................................................................. 37
3.2 Built-in Data Types ................................................................................. 38
  3.2.1 Registering Built-in Types ................................................................. 39
  3.2.2 Creating Topics for Built-in Types .................................................. 39
  3.2.3 String Built-in Type .......................................................................... 41
  3.2.4 KeyedString Built-in Type .............................................................. 46
  3.2.5 Octets Built-in Type ......................................................................... 54
  3.2.6 KeyedOctets Built-in Type .............................................................. 60
  3.2.7 Managing Memory for Built-in Types .............................................. 69
  3.2.8 Type Codes for Built-in Types ......................................................... 74
3.3 Creating User Data Types with IDL ...................................................... 75
  3.3.1 Variable-Length Types .................................................................... 77
  3.3.2 Value Types ..................................................................................... 78
  3.3.3 Type Codes ..................................................................................... 79
  3.3.4 Translations for IDL Types ............................................................... 79
  3.3.5 Escaped Identifiers .......................................................................... 108
  3.3.6 Namespaces In IDL Files ................................................................. 108
  3.3.7 Referring to Other IDL Files ............................................................ 111
  3.3.8 Preprocessor Directives .................................................................. 112
  3.3.9 Using Builtin Annotations ............................................................... 112
3.4 Creating User Data Types with Extensible Markup Language (XML) .... 121
3.5 Creating User Data Types with XML Schemas (XSD) ............................ 131
  3.5.1 Primitive Types ............................................................................. 151
3.6 Using RTI Code Generator (rtiddsgen) .................................................. 152
3.7 Using Generated Types without Connext DDS (Standalone) ................. 152
  3.7.1 Using Standalone Types in C ........................................................... 152
  3.7.2 Using Standalone Types in C++ ..................................................... 153
  3.7.3 Standalone Types in Java ............................................................... 154
3.8 Interacting Dynamically with User Data Types ...................................... 154
  3.8.1 Type Schemas and TypeCode Objects .......................................... 154
  3.8.2 Defining New Types ....................................................................... 154
  3.8.3 Sending Only a Few Fields ............................................................. 156
  3.8.4 Sending Type Information on the Network ..................................... 157
### 3.9 Working with DDS Data Samples

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.9.1 Objects of Concrete Types</td>
<td>158</td>
</tr>
<tr>
<td>3.9.2 Objects of Dynamically Defined Types</td>
<td>161</td>
</tr>
<tr>
<td>3.9.3 Serializing and Deserializing Data Samples</td>
<td>163</td>
</tr>
<tr>
<td>3.9.4 Accessing the Discriminator Value in a Union</td>
<td>163</td>
</tr>
</tbody>
</table>

### 3.10 Data Sample Serialization Limits

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.10</td>
<td>164</td>
</tr>
</tbody>
</table>

---

### Chapter 4 DDS Entities

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Common Operations for All DDS Entities</td>
<td>166</td>
</tr>
<tr>
<td>4.1.1 Creating and Deleting DDS Entities</td>
<td>167</td>
</tr>
<tr>
<td>4.1.2 Enabling DDS Entities</td>
<td>168</td>
</tr>
<tr>
<td>4.1.3 Getting an Entity’s Instance Handle</td>
<td>171</td>
</tr>
<tr>
<td>4.1.4 Getting Status and Status Changes</td>
<td>171</td>
</tr>
<tr>
<td>4.1.5 Getting and Setting Listeners</td>
<td>171</td>
</tr>
<tr>
<td>4.1.6 Getting the StatusCondition</td>
<td>172</td>
</tr>
<tr>
<td>4.1.7 Getting, Setting, and Comparing QosPolicies</td>
<td>172</td>
</tr>
<tr>
<td>4.2 QosPolicies</td>
<td>176</td>
</tr>
<tr>
<td>4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property</td>
<td>180</td>
</tr>
<tr>
<td>4.2.2 Special QosPolicy Handling Considerations for C</td>
<td>182</td>
</tr>
<tr>
<td>4.3 Statuses</td>
<td>183</td>
</tr>
<tr>
<td>4.3.1 Types of Communication Status</td>
<td>183</td>
</tr>
<tr>
<td>4.3.2 Special Status-Handling Considerations for C</td>
<td>188</td>
</tr>
<tr>
<td>4.4 Listeners</td>
<td>189</td>
</tr>
<tr>
<td>4.4.1 Types of Listeners</td>
<td>190</td>
</tr>
<tr>
<td>4.4.2 Creating and Deleting Listeners</td>
<td>192</td>
</tr>
<tr>
<td>4.4.3 Special Considerations for Listeners in C</td>
<td>193</td>
</tr>
<tr>
<td>4.4.4 Special Considerations for Listeners in Modern C++</td>
<td>193</td>
</tr>
<tr>
<td>4.4.5 Hierarchical Processing of Listeners</td>
<td>194</td>
</tr>
<tr>
<td>4.4.6 Operations Allowed within Listener Callbacks</td>
<td>196</td>
</tr>
<tr>
<td>4.4.7 Best Practices with Listeners</td>
<td>196</td>
</tr>
<tr>
<td>4.5 Exclusive Areas (EAs)</td>
<td>197</td>
</tr>
<tr>
<td>4.5.1 Restricted Operations in Listener Callbacks</td>
<td>200</td>
</tr>
<tr>
<td>4.6 Conditions and WaitSets</td>
<td>202</td>
</tr>
<tr>
<td>4.6.1 Creating and Deleting WaitSets</td>
<td>203</td>
</tr>
<tr>
<td>4.6.2 WaitSet Operations</td>
<td>204</td>
</tr>
<tr>
<td>4.6.3 Waiting for Conditions</td>
<td>205</td>
</tr>
<tr>
<td>4.6.4 Processing Triggered Conditions—What to do when Wait() Returns</td>
<td>207</td>
</tr>
</tbody>
</table>
Chapter 5 Working with Topics

5.1 Topics ................................................................. 215
  5.1.1 Creating Topics ............................................... 217
  5.1.2 Deleting Topics ............................................... 219
  5.1.3 Setting Topic QosPolicies ................................. 219
  5.1.4 Copying QoS From a Topic to a DataWriter or DataReader ........................................ 222
  5.1.5 Setting Up TopicListeners ................................ 223
  5.1.6 Navigating Relationships Among Entities .......... 223

5.2 Topic QosPolicies .................................................. 224
  5.2.1 TOPIC_DATA QosPolicy .................................. 224

5.3 Status Indicator for Topics ...................................... 226
  5.3.1 INCONSISTENT_TOPIC Status ......................... 226

5.4 ContentFilteredTopics ........................................... 227
  5.4.1 Overview ......................................................... 227
  5.4.2 Where Filtering is Applied—Publishing vs. Subscribing Side ........................................ 228
  5.4.3 Creating ContentFilteredTopics ....................... 229
  5.4.4 Deleting ContentFilteredTopics ....................... 233
  5.4.5 Using a ContentFilteredTopic ................................ 233
  5.4.6 SQL Filter Expression Notation ...................... 236
  5.4.7 STRINGMATCH Filter Expression Notation ........ 245
  5.4.8 Character Encoding ......................................... 246
  5.4.9 Unicode Normalization .................................... 247
  5.4.10 Custom Content Filters .................................. 248

Chapter 6 Working with Instances

6.1 Instance States ..................................................... 258
  6.1.1 ALIVE Details ............................................... 259
  6.1.2 NOT_ALIVE_DISPOSED Details ....................... 259
  6.1.3 NOT_ALIVE_NO_WRITERS Details .................... 261
  6.1.4 Transitions between NOT_ALIVE States .............. 262

6.2 Instance Memory Management .................................. 264

6.3 QoS Configuration and Instances ............................. 266
7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) ..............................................350
7.4.4 GROUP_DATA QosPolicy .................................................................353
7.4.5 PARTITION QosPolicy .................................................................356
7.4.6 PRESENTATION QosPolicy ...............................................................362
7.5 DataWriter QosPolicies ........................................................................369
  7.5.1 AVAILABILITY QosPolicy (DDS Extension) ........................................370
  7.5.2 BATCH QosPolicy (DDS Extension) ..................................................374
  7.5.3 DATA_REPRESENTATION QosPolicy ..............................................380
  7.5.4 DATATAG QosPolicy .................................................................388
  7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) .....................389
  7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) .......401
  7.5.7 DEADLINE QosPolicy ...................................................................405
  7.5.8 DESTINATION_ORDER QosPolicy ...................................................408
  7.5.9 DURABILITY QosPolicy ...............................................................411
  7.5.10 DURABILITY SERVICE QosPolicy ..............................................416
  7.5.11 ENTITY_NAME QosPolicy (DDS Extension) ..................................418
  7.5.12 HISTORY QosPolicy ..................................................................420
  7.5.13 LATENCYBUDGET QoS Policy .....................................................425
  7.5.14 LIFESPAN QoS Policy ...............................................................425
  7.5.15 LIVELINESS QosPolicy ..............................................................427
  7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) ...............................432
  7.5.17 OWNERSHIP QosPolicy ............................................................434
  7.5.18 OWNERSHIP_STRENGTH QosPolicy ...........................................438
  7.5.19 PROPERTY QosPolicy (DDS Extension) .......................................439
  7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) ...............................444
  7.5.21 RELIABILITY QosPolicy ............................................................447
  7.5.22 RESOURCE_LIMITS QosPolicy ..................................................451
  7.5.23 SERVICE QosPolicy (DDS Extension) .........................................454
  7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) ...............455
  7.5.25 TRANSFER_MODE QosPolicy ......................................................457
  7.5.26 TRANSPORT_PRIORITY QosPolicy .............................................458
  7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) ...................459
  7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) ......................460
  7.5.29 TYPESUPPORT QosPolicy (DDS Extension) ..................................464
  7.5.30 USER_DATA QosPolicy .............................................................465
  7.5.31 WRITER_DATA_LIFECYCLE QoS Policy .......................................467
7.6 FlowControllers (DDS Extension) .......................................................... 470
  7.6.1 Flow Controller Scheduling Policies .............................................. 472
  7.6.2 Managing Fast DataWriters When Using a FlowController ............. 473
  7.6.3 Token Bucket Properties .............................................................. 474
  7.6.4 Prioritized DDS Samples .............................................................. 476
  7.6.5 Creating and Configuring Custom FlowControllers with Property QoS .. 479
  7.6.6 Creating and Deleting FlowControllers .......................................... 481
  7.6.7 Getting/Setting Default FlowController Properties ......................... 482
  7.6.8 Getting/Setting Properties for a Specific FlowController ............... 482
  7.6.9 Adding an External Trigger ......................................................... 483
  7.6.10 Other FlowController Operations .............................................. 483

Chapter 8 Receiving Data

8.1 Preview: Steps to Receiving Data ...................................................... 484
8.2 Subscribers ...................................................................................... 486
  8.2.1 Creating Subscribers Explicitly vs. Implicitly ................................. 490
  8.2.2 Creating Subscribers ................................................................. 491
  8.2.3 Deleting Subscribers ................................................................. 492
  8.2.4 Setting Subscriber QosPolicies .................................................. 493
  8.2.5 Beginning and Ending Group-Ordered Access ............................... 499
  8.2.6 Setting Up SubscriberListeners .................................................. 500
  8.2.7 Getting DataReaders with Specific DDS Samples ......................... 502
  8.2.8 Finding a Subscriber’s Related Entities ...................................... 502
  8.2.9 Statuses for Subscribers ............................................................ 503

8.3 DataReaders .................................................................................... 504
  8.3.1 Creating DataReaders ................................................................. 510
  8.3.2 Getting All DataReaders .............................................................. 512
  8.3.3 Deleting DataReaders ................................................................. 512
  8.3.4 Setting Up DataReaderListeners ................................................. 512
  8.3.5 Checking DataReader Status and StatusConditions ....................... 514
  8.3.6 Waiting for Historical Data ......................................................... 515
  8.3.7 Statuses for DataReaders ............................................................ 516
  8.3.8 Accessing and Managing Instances (Working with Keyed Data Types) 534
  8.3.9 Setting DataReader QosPolicies ................................................. 543
  8.3.10 Navigating Relationships Among Entities .................................. 551

8.4 Using DataReaders to Access Data (Read & Take) .............................. 553
  8.4.1 Using a Type-Specific DataReader (FooDataReader) ....................... 553
8.4.2 Loaning and Returning Data and SampleInfo Sequences ........................................553
8.4.3 Accessing DDS Data Samples with Read or Take ...............................................555
8.4.4 Acknowledging DDS Samples ..............................................................................563
8.4.5 The Sequence Data Structure ..............................................................................563
8.4.6 The SampleInfo Structure ..................................................................................565
8.5 Subscriber QoS Policies .........................................................................................570
8.6 DataReader QoS Policies .........................................................................................570
   8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) ...................................570
   8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) ......................576
   8.6.3 READER_DATA_LIFECYCLE QoS Policy ...............................................................586
   8.6.4 TIME_BASED_FILTER QosPolicy .......................................................................588
   8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) .......................................591
   8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy ..............................................594

Chapter 9 Working with DDS Domains

9.1 Fundamentals of DDS Domains and DomainParticipants ........................................600
9.2 DomainParticipantFactory ....................................................................................602
   9.2.1 Setting DomainParticipantFactory QoS Policies ...............................................605
   9.2.2 Getting and Setting Default QoS for DomainParticipants ...................................607
   9.2.3 Freeing Resources Used by the DomainParticipantFactory .................................607
   9.2.4 Looking Up DomainParticipants .......................................................................608
   9.2.5 Getting QoS Values from a QoS Profile ...............................................................608
9.3 DomainParticipants ...............................................................................................609
   9.3.1 Creating a DomainParticipant ........................................................................615
   9.3.2 Deleting DomainParticipants ...........................................................................617
   9.3.3 Deleting Contained Entities ...............................................................................617
   9.3.4 Choosing a Domain ID and Creating Multiple DDS Domains ...............................618
   9.3.5 Choosing a Domain Tag ....................................................................................619
   9.3.6 Setting Up DomainParticipantListeners ............................................................620
   9.3.7 Setting DomainParticipant QoS Policies .............................................................622
   9.3.8 Looking up Topic Descriptions .........................................................................628
   9.3.9 Finding a Topic ..................................................................................................628
   9.3.10 Getting the Implicit Publisher or Subscriber ......................................................629
   9.3.11 Asserting Liveliness ........................................................................................630
   9.3.12 Learning about Discovered DomainParticipants ..............................................630
   9.3.13 Learning about Discovered Topics ....................................................................630
   9.3.14 Getting Participant Protocol Status ....................................................................631
11.1.2 Reliable Delivery Model ................................................................. 689
11.2 Overview of the Reliable Protocol ....................................................... 690
11.3 Using QosPolicies to Tune the Reliable Protocol ..................................... 694
  11.3.1 Enabling Reliability ................................................................. 696
  11.3.2 Tuning Queue Sizes and Other Resource Limits .................................. 696
  11.3.3 Controlling Queue Depth with the History QosPolicy ............................ 703
  11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy .... 704
  11.3.5 Avoiding Message Storms with DataReaderProtocol QosPolicy .......... 712
  11.3.6 Resending DDS Samples to Late-Joiners with the Durability QosPolicy ...... 712
  11.3.7 Use Cases ............................................................................. 713
11.4 Auto Throttling for DataWriter Performance—Experimental Feature ............. 725

Chapter 12 Collaborative DataWriters
12.1 Collaborative DataWriters Use Cases .................................................... 728
  12.2 DDS Sample Combination (Synchronization) Process in a DataReader .......... 729
12.3 Configuring Collaborative DataWriters ................................................. 730
  12.3.1 Associating Virtual GUIDs with DDS Data Samples ......................... 730
  12.3.2 Associating Virtual Sequence Numbers with DDS Data Samples .......... 730
  12.3.3 Specifying which DataWriters will Deliver DDS Samples to the DataReader from a Logical Data Source ........................................ 730
  12.3.4 Specifying How Long to Wait for a Missing DDS Sample .................... 730
12.4 Collaborative DataWriters and Persistence Service ................................. 731

Chapter 13 Mechanisms for Achieving Information Durability and Persistence
13.1 Introduction .................................................................................. 732
  13.1.1 Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History) ................................................................. 733
  13.1.2 Scenario 2: DataReader Restarts While DataWriter Stays Up (Durable Reader State) ................................................................. 735
  13.1.3 Scenario 3. DataReader Joins after DataWriter Leaves Domain (Durable Data) ................................................................. 736
13.2 Durability and Persistence Based on Virtual GUIDs .................................. 737
13.3 Durable Writer History .................................................................... 739
  13.3.1 Durable Writer History Use Case ............................................... 740
  13.3.2 How To Configure Durable Writer History ...................................... 741
13.4 Durable Reader State .................................................................... 744
  13.4.1 Durable Reader State With Protocol Acknowledgment ..................... 744
  13.4.2 Durable Reader State with Application Acknowledgment .................. 746
  13.4.3 Durable Reader State Use Case ............................................... 746
  13.4.4 How To Configure a DataReader for Durable Reader State ................ 747
13.5 Data Durability .............................................................................. 749
13.5.1 RTI Persistence Service .......................................................... 749

Chapter 14 Guaranteed Delivery of Data

14.1 Introduction .................................................................................. 752
  14.1.1 Identifying the Required Consumers of Information ................. 753
  14.1.2 Ensuring Consumer Applications Process the Data Successfully .... 755
  14.1.3 Ensuring Information is Available to Late-Joining Applications .... 756

14.2 Scenarios ...................................................................................... 757
  14.2.1 Scenario 1: Guaranteed Delivery to a-priori Known Subscribers .... 757
  14.2.2 Scenario 2: Surviving a Writer Restart when Delivering DDS Samples to a priori Known Subscribers .... 759
  14.2.3 Scenario 3: Delivery Guaranteed by Persistence Service (Store and Forward) to a priori Known Subscribers ............................................................................. 760

Chapter 15 Discovery

15.1 What is Discovery? ......................................................................... 765
  15.1.1 Simple Participant Discovery ..................................................... 765
  15.1.2 Simple Endpoint Discovery ....................................................... 766

15.2 Configuring the Peers List Used in Discovery ............................... 767
  15.2.1 Peer Descriptor Format ............................................................ 769
  15.2.2 NDDS_DISCOVERY_PEERS Environment Variable Format ........ 771
  15.2.3 NDDS_DISCOVERY_PEERS File Format .................................. 772

15.3 Discovery Implementation ............................................................ 773
  15.3.1 Participant Discovery ............................................................... 774
  15.3.2 Endpoint Discovery ................................................................. 783
  15.3.3 Discovery Traffic Summary ..................................................... 787
  15.3.4 Discovery-Related QoS ............................................................ 788

15.4 Debugging Discovery .................................................................... 789

15.5 Ports Used for Discovery ............................................................... 791
  15.5.1 Inbound Ports for Meta-Traffic ............................................... 793
  15.5.2 Inbound Ports for User Traffic ................................................. 793
  15.5.3 Automatic Selection of participant_id and Port Reservation ....... 794
  15.5.4 Tuning domain_id_gain and participant_id_gain ....................... 794

Chapter 16 Transport Plugins

16.1 Built-in Transport Plugins ............................................................ 796
16.2 Extension Transport Plugins .......................................................... 797
16.3 The NDDSTransportSupport Class ............................................... 798
16.4 Explicitly Creating Built-in Transport Plugin Instances ................... 798
16.5 Setting Built-in Transport Properties of Default Transport Instance—get/set_builtin_transport_properties() 799
Chapter 19 Configuring Quality of Service (QoS) with XML

19.3.4 Topic Filters ................................................................. 870
19.3.5 QoS Profiles with a Single QoS ....................................... 874
19.4 Tags for Configuring QoS with XML ..................................... 874
19.4.1 QoS Policies ............................................................... 875
19.4.2 Sequences ................................................................. 875
19.4.3 Arrays ..................................................................... 878
19.4.4 Enumeration Values ...................................................... 879
19.4.5 Time Values (Durations) ............................................... 879
19.4.6 Transport Properties ..................................................... 880
19.4.7 Thread Settings .......................................................... 882
19.4.8 Entity Names .............................................................. 882
19.5 How to Load XML-Specified QoS Settings ............................... 882
19.5.1 Loading, Reloading and Unloading Profiles........................... 884
19.6 XML File Syntax ............................................................. 885
19.6.1 Using Environment Variables in XML ................................. 886
19.6.2 Using Special Characters in XML .................................... 887
19.6.3 Specifying Fully Qualified Names in XML ............................ 888
19.7 XML String Syntax .......................................................... 888
19.8 URL Groups (Loading Redundant Locations) ............................. 889
19.9 How the XML is Validated .................................................. 889
19.9.1 Validation at Run-Time ............................................... 889
19.9.2 XML File Validation During Editing ................................ 890
19.10 Using QoS Profiles in Your Connext DDS Application ............. 891
19.10.1 Retrieving a List of Available Libraries ............................. 894
19.10.2 Retrieving a List of Available QoS Profiles ......................... 895
19.11 Configuring Logging Via XML ........................................... 895

Chapter 20 Multi-channel DataWriters

20.1 What is a Multi-channel DataWriter? .................................... 897
20.2 How to Configure a Multi-channel DataWriter ......................... 900
20.2.1 Limitations .............................................................. 901
20.3 Multi-Channel Configuration on the Reader Side ....................... 901
20.4 Where Does the Filtering Occur? ....................................... 903
20.4.1 Filtering at the DataWriter .......................................... 903
20.4.2 Filtering at the DataReader ......................................... 903
20.4.3 Filtering on the Network Hardware ................................ 904
20.5 Fault Tolerance and Redundancy ........................................ 904
20.6 Reliability with Multi-Channel DataWriters ................................................................. 905
20.6.1 Reliable Delivery ........................................................................................................... 905
20.6.2 Reliable Protocol Considerations ............................................................................... 905
20.7 Performance Considerations ......................................................................................... 906
20.7.1 Network-Switch Filtering ......................................................................................... 906
20.7.2 DataWriter and DataReader Filtering ....................................................................... 906

Chapter 21 Connext DDS Threading Model
21.1 Database Thread ........................................................................................................... 909
21.2 Event Thread ................................................................................................................ 910
21.3 Receive Threads ............................................................................................................ 911
21.4 Exclusive Areas, RTI Connext DDS Threads, and User Listeners .................................. 913
21.5 Controlling CPU Core Affinity for RTI Threads ........................................................... 913
21.6 Configuring Thread Settings with XML ....................................................................... 914
21.7 User-Managed Threads ................................................................................................ 915
21.8 Unregistering Threads .................................................................................................. 916
21.9 Identifying Threads Used by Connext DDS ................................................................. 916
21.9.1 Checking Thread Names at the OS Level .................................................................... 917
21.9.2 Checking Thread Names from the Call Stack .............................................................. 923
21.9.3 Checking Thread Names Using the Worker’s Name .................................................... 924

Chapter 22 DDS Sample and Instance Memory Management
22.1 DDS Sample Memory Management for DataWriters ..................................................... 926
22.1.1 Memory Management without Batching .................................................................... 927
22.1.2 Memory Management with Batching ........................................................................ 929
22.1.3 Writer-Side Memory Management when Using Java ................................................ 931
22.1.4 Writer-Side Memory Management when Working with Large Data ......................... 931
22.2 DDS Sample Memory Management for DataReaders ................................................... 933
22.2.1 Memory Management for DataReaders Using Generated Type-Plugins .................... 934
22.2.2 Reader-Side Memory Management when Using Java ............................................... 935
22.2.3 Memory Management for DynamicData DataReaders ............................................. 936
22.2.4 Memory Management for Fragmented DDS Samples ................................................ 938
22.2.5 Reader-Side Memory Management when Working with Large Data ......................... 938
22.3 Instance Memory Management for DataWriters .......................................................... 940
22.4 Instance Memory Management for DataReaders ......................................................... 940

Chapter 23 Sending Large Data
23.1 Reducing Latency .......................................................................................................... 944
23.1.1 Use Cases ................................................................................................................ 945
28.2.3 Summary of Client Operations ................................................................. 1030
28.3 Accessing Underlying DataWriters and DataReaders ................................. 1032
28.4 Generating RPC Code from IDL using RTI Code Generator ........................... 1032

Part 5: RTI Real-Time WAN Transport ............................................................... 1034

Chapter 29 Introduction to Real-Time WAN Transport

29.1 Key Terms ........................................................................................................... 1036
  29.1.1 Basic Terms .................................................................................................... 1036
  29.1.2 IP Address Types ......................................................................................... 1036
  29.1.3 Locators ........................................................................................................ 1036
  29.1.4 WAN Ecosystem .......................................................................................... 1037

Chapter 30 Transport Capabilities

30.1 NAT Traversal .................................................................................................... 1038
  30.1.1 NAT Kinds ..................................................................................................... 1039
  30.1.2 Identifying the NAT Type ............................................................................. 1041
  30.1.3 NAT Bindings ............................................................................................... 1041
  30.1.4 NAT Bindings Expiration ............................................................................. 1042
  30.1.5 NAT Hairpinning ......................................................................................... 1043
  30.2 IP Mobility ....................................................................................................... 1043

Chapter 31 Communication Scenarios

31.1 Peer-to-Peer Communication with a Participant that has a Public Address ......... 1045
  31.1.1 External Participant Configuration: Sub-Scenario 1 .................................... 1047
  31.1.2 External Participant Configuration: Sub-Scenario 2 .................................... 1047
  31.1.3 Internal Participants Configuration ............................................................... 1048
  31.2 Peer-to-Peer Communication with Participants behind Cone NATs Using Cloud Discovery Service ................................................................. 1048
    31.2.1 Internal Participants Configuration ............................................................... 1050
    31.2.2 Cloud Discovery Service Configuration ..................................................... 1050

Chapter 32 Deployment Scenarios

32.1 Edge-to-Data Center Deployment Scenario ..................................................... 1052
  32.1.1 Data Center Routing Service Configuration ................................................ 1054
  32.1.2 In-Vehicle Routing Service Configuration .................................................. 1056
  32.2 Relayed Edge-to-Edge Deployment Scenario ................................................ 1057
    32.2.1 Data Center Routing Service Configuration ................................................ 1060
    32.2.2 Webinar Application Configuration ............................................................. 1061
  32.3 Peer-to-Peer, Edge-to-Edge Deployment Scenario .......................................... 1061
    32.3.1 Video Connext Application Configuration .................................................. 1063
    32.3.2 Cloud Discovery Service Configuration ..................................................... 1063
Chapter 33 Enabling Real-Time WAN Transport

33.1 Dynamically Loading the Real-Time WAN Transport ........................................... 1064
33.2 Linking the Real-Time WAN Transport against your Application ...................... 1065

Chapter 34 Transport Initial Peers .............................................................................. 1066

Chapter 35 Transport Configuration

35.1 Setting Real-Time WAN Transport Properties .................................................... 1068
35.2 Managing UDP Ports Used for Communication ................................................ 1078
  35.2.1 Receiving Data ............................................................................................... 1078
  35.2.2 Configuring the Transport to Use a Single Port for an Internal Participant behind a NAT .... 1080
  35.2.3 Configuring the Transport to Segregate Traffic for a Topic in its own Port ............ 1082
  35.2.4 Sending Data ............................................................................................... 1083
35.3 Disabling IP Fragmentation for Real-Time WAN Transport .................................... 1084
  35.3.1 RTPS Overhead ......................................................................................... 1085

Chapter 36 Security ..................................................................................................... 1086

Chapter 37 Advanced Concepts

37.1 Transport Locators .............................................................................................. 1087
37.2 Binding Ping Messages ....................................................................................... 1088
37.3 Communication Establishment Protocol for Peer-to-Peer Communication with Participants behind Cone NATs ................................................................. 1089
37.4 Communication Establishment Protocol for Peer-to-Peer Communication with a Participant that has a Public Address .............................................................. 1092

Chapter 38 Transport Debugging

38.1 Debugging Peer-to-Peer Communication with a Participant that has a Public Address ........... 1095
38.2 Peer-to-Peer Communication with Participants behind Cone NATs ....................... 1098

Chapter 39 Tools Integration ..................................................................................... 1102

Chapter 40 Troubleshooting

40.1 Communication Stops Working after Application Transitions to Different Network ...... 1104
40.2 Communication not Established after Changing Cloud Discovery Service <receiver_port> .... 1104
40.3 Communication not Established even though Transport Settings are Set Correctly ........ 1105
40.4 "serialize buffer too small" Error ........................................................................ 1106
40.5 Slow Discovery using Cloud Discovery Service .................................................... 1107

Part 6: RTI Secure WAN Transport .......................................................................... 1109

Chapter 41 Introduction to Secure WAN Transport

41.1 WAN Traversal via UDP Hole-Punching ............................................................ 1111
  41.1.1 Protocol Details ......................................................................................... 1112
41.2 WAN Locators .................................................................................................... 1115
41.3 Datagram Transport-Layer Security (DTLS) ...................................................... 1116
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>41.3.1 Security Model</td>
<td>1117</td>
</tr>
<tr>
<td>41.3.2 Liveliness Mechanism</td>
<td>1118</td>
</tr>
<tr>
<td>41.4 Certificate Support</td>
<td>1118</td>
</tr>
<tr>
<td>41.5 License Issues</td>
<td>1119</td>
</tr>
<tr>
<td><strong>Chapter 42 Configuring RTI Secure WAN Transport</strong></td>
<td></td>
</tr>
<tr>
<td>42.1 Example Applications</td>
<td>1123</td>
</tr>
<tr>
<td>42.2 Setting Up a Transport with the Property QoS</td>
<td>1123</td>
</tr>
<tr>
<td>42.3 WAN Transport Properties</td>
<td>1125</td>
</tr>
<tr>
<td>42.4 Secure Transport Properties</td>
<td>1133</td>
</tr>
<tr>
<td>42.5 Explicitly Instantiating a WAN or Secure Transport Plugin</td>
<td>1139</td>
</tr>
<tr>
<td>42.5.1 Additional Header Files and Include Directories</td>
<td>1140</td>
</tr>
<tr>
<td>42.5.2 Additional Libraries</td>
<td>1140</td>
</tr>
<tr>
<td>42.5.3 Compiler Flags</td>
<td>1141</td>
</tr>
<tr>
<td><strong>Part 7: RTI TCP Transport</strong></td>
<td>1142</td>
</tr>
<tr>
<td><strong>Chapter 43 TCP Communication Scenarios</strong></td>
<td></td>
</tr>
<tr>
<td>43.1 Communication Within a Single LAN</td>
<td>1143</td>
</tr>
<tr>
<td>43.2 Symmetric Communication Across NATs</td>
<td>1144</td>
</tr>
<tr>
<td>43.3 Asymmetric Communication Across NATs</td>
<td>1145</td>
</tr>
<tr>
<td><strong>Chapter 44 Configuring the TCP Transport</strong></td>
<td></td>
</tr>
<tr>
<td>44.1 Choosing a Transport Mode</td>
<td>1148</td>
</tr>
<tr>
<td>44.2 Explicitly Instantiating the TCP Transport Plugin</td>
<td>1149</td>
</tr>
<tr>
<td>44.2.1 Additional Header Files and Include Directories</td>
<td>1150</td>
</tr>
<tr>
<td>44.2.2 Additional Libraries and Compiler Flags</td>
<td>1150</td>
</tr>
<tr>
<td>44.3 Configuring the TCP Transport with the Property QosPolicy</td>
<td>1151</td>
</tr>
<tr>
<td>44.3.1 Configuring the TCP Transport to be Loaded Statically</td>
<td>1153</td>
</tr>
<tr>
<td>44.3.2 Loading TLS Support Libraries Statically</td>
<td>1154</td>
</tr>
<tr>
<td>44.4 Setting the Initial Peers</td>
<td>1154</td>
</tr>
<tr>
<td>44.5 RTPS Locator Format</td>
<td>1155</td>
</tr>
<tr>
<td>44.6 Support for External Hardware Load Balancers in TCP Transport Plugin</td>
<td>1156</td>
</tr>
<tr>
<td>44.6.1 Session-ID Messages</td>
<td>1158</td>
</tr>
<tr>
<td>44.7 TCP/TLS Transport Properties</td>
<td>1158</td>
</tr>
<tr>
<td>44.7.1 Connection Liveliness</td>
<td>1177</td>
</tr>
<tr>
<td><strong>Part 8: RTI Persistence Service</strong></td>
<td>1179</td>
</tr>
<tr>
<td><strong>Chapter 45 Introduction to RTI Persistence Service</strong></td>
<td>1180</td>
</tr>
<tr>
<td><strong>Chapter 46 Configuring Persistence Service</strong></td>
<td></td>
</tr>
<tr>
<td>46.1 How to Load the Persistence Service XML Configuration</td>
<td>1182</td>
</tr>
</tbody>
</table>

xxiv
Chapter 46 Configuring the Persistence Service

46.2 XML Configuration File
    46.2.1 Configuration File Syntax
    46.2.2 XML Validation

46.3 QoS Configuration

46.4 Configuring the Persistence Service Application

46.5 Configuring Remote Administration

46.6 Configuring Persistent Storage

46.7 Configuring Participants

46.8 Creating Persistence Groups
    46.8.1 QoSs
    46.8.2 DurabilityService QoS Policy
    46.8.3 Sharing a Publisher/Subscriber
    46.8.4 Sharing a Database Connection
    46.8.5 Memory Management

46.9 Configuring Durable Subscriptions in Persistence Service
    46.9.1 DDS Sample Memory Management With Durable Subscriptions

46.10 Synchronizing of Persistence Service Instances

46.11 Enabling RTI Distributed Logger in Persistence Service

46.12 Enabling RTI Monitoring Library in Persistence Service

46.13 Support for Extensible Types
    46.13.1 TypeConsistencyEnforcementQosPolicy Integration
    46.13.2 DataRepresentationQosPolicy Integration

46.14 TCP Transport Support in Persistence Service

Chapter 47 Running RTI Persistence Service

47.1 Starting Persistence Service

47.2 Stopping Persistence Service

Chapter 48 Administering Persistence Service from a Remote Location

48.1 Enabling Remote Administration

48.2 Remote Commands
    48.2.1 start
    48.2.2 stop
    48.2.3 shutdown
    48.2.4 status

48.3 Accessing Persistence Service from a Connext DDS Application

Chapter 49 Advanced Persistence Service Scenarios

49.1 Scenario: Load-balanced Persistence Services
About this Document

Paths Mentioned in Documentation

The documentation refers to:

- `<NDDSHOME>`

  This refers to the installation directory for RTI® Connext® DDS. The default installation paths are:
  - macOS® systems:
    `/Applications/rti_connext_dds-6.1.1`
  - Linux systems, non-root user:
    `/home/<your user name>/rti_connext_dds-6.1.1`
  - Linux systems, root user:
    `/opt/rti_connext_dds-6.1.1`
  - Windows® systems, user without Administrator privileges:
    `<your home directory>\rti_connext_dds-6.1.1`
  - Windows systems, user with Administrator privileges:
    `C:\Program Files\rti_connext_dds-6.1.1`

You may also see `$NDDSHOME` or `%NDDSHOME%`, which refers to an environment variable set to the installation path.

Wherever you see `<NDDSHOME>` used in a path, replace it with your installation path.
Note for Windows Users: When using a command prompt to enter a command that includes the path C:\Program Files (or any directory name that has a space), enclose the path in quotation marks. For example:

"C:\Program Files\rti_connexx_dds-6.1.1\bin\rtiddsgen"

Or if you have defined the NDDSHOME environment variable:

"%NDDSHOME%\bin\rtiddsgen"

- <path to examples>

By default, examples are copied into your home directory the first time you run RTI Launcher or any script in <NDDSHOME>/bin. This document refers to the location of the copied examples as <path to examples>.

Wherever you see <path to examples>, replace it with the appropriate path.

Default path to the examples:

- macOS systems: /Users/<your user name>/rti_workspace/6.1.1/examples
- Linux systems: /home/<your user name>/rti_workspace/6.1.1/examples
- Windows systems: <your Windows documents folder>\rti_workspace\6.1.1\examples

Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 10, the folder is C:\Users\<your user name>\Documents.

Note: You can specify a different location for rti_workspace. You can also specify that you do not want the examples copied to the workspace. For details, see Controlling Location for RTI Workspace and Copying of Examples in the RTI Connext DDS Installation Guide.

Programming Language Conventions

The terminology and example code in this manual assume you are using Traditional C++ without namespace support.

C, Modern C++, C++/CLI, C#, and Java APIs are also available; they are fully described in the API Reference HTML documentation. (Note: the Modern C++ API is not available for all platforms, check the RTI Connext DDS Core Libraries Platform Notes to see if it is available for your platform.)

Namespace support in Traditional C++, C++/CLI, and C# is also available; see the API Reference HTML documentation (from the Modules page, select Using DDS:: Namespace) for details. In the Modern C++ API all types, constants and functions are always in namespaces.

Traditional vs. Modern C++

Connext DDS provides two different C++ APIs, which we refer to as the "Traditional C++" and "Modern C++" APIs. They provide substantially different programming paradigms and patterns. The Traditional
API could be considered as simply "C with classes," while the Modern API incorporates modern C++ techniques, most notably:

- Generic programming
- Integration with the standard library
- Automatic object lifecycle management, providing full value types and reference types
- C++11 support, such as move operations, initializer lists, and support for range for-loops.

These different programming styles make the Modern C++ API differ significantly with respect to the other language APIs in several aspects; to name a few:

- 3.3 Creating User Data Types with IDL on page 75
- 3.8 Interacting Dynamically with User Data Types on page 154
- 3.9 Working with DDS Data Samples on page 158
- 4.1.1 Creating and Deleting DDS Entities on page 167
- 8.4 Using DataReaders to Access Data (Read & Take) on page 553
- QoS policies and QoS management
- Naming conventions

This manual points out these kinds of differences whenever they are substantial.

**Extensions to the DDS Standard**

Connext DDS implements the DDS Standard published by the OMG. It also includes features that are extensions to DDS. These include additional Quality of Service parameters, function calls, structure fields, etc.

Extensions also include product-specific APIs that complement the DDS API. These include APIs to create and use transport plug-ins, and APIs to control the verbosity and logging capabilities. These APIs are prefixed with NDDS, such as `NDDSTransportSupport::register_transport()`.

**Environment Variables**

Connext DDS documentation refers to path names that have been customized during installation. NDDSHOME refers to the installation directory of Connext DDS.

**Names of Supported Platforms**

Connext DDS runs on several different target platforms. To support this vast array of platforms, Connext DDS separates the executable, library, and object files for each platform into individual directories.
Each platform name has four parts: hardware architecture, operating system, operating system version and compiler. For example, **x64Linux2.6gcc4.4.5** is the directory that contains files specific to Linux® version 2.6 for the x64 Intel processor, compiled with gcc version 4.4.5.

For a full list of supported platforms, see the [RTI Connext DDS Core Libraries Platform Notes](#).

**Additional Resources**

The details of each API (such as function parameters, return values, etc.) and examples are in the API Reference HTML documentation. In case of discrepancies between the information in this document and the API Reference HTML documentation, the latter should be considered more up-to-date.
Part 1: Welcome to RTI Connext DDS

RTI® Connext® DDS solutions provide a flexible connectivity software framework for integrating data sources of all types. At its core is the world's leading ultra-high performance, distributed networking Data-abus. It connects data within applications as well as across devices, systems and networks. Connext DDS also delivers large data sets with microsecond performance and granular quality-of-service control. Connext DDS is a standards-based, open architecture that connects devices from deeply embedded real-time platforms to enterprise servers across a variety of networks.

Part 1 introduces the general concepts behind data-centric publish-subscribe communications and provides a brief tour of Connext DDS.

- Overview (Chapter 1 on page 6)
- Data-Centric Publish-Subscribe Communications (Chapter 2 on page 14)
Chapter 1 Overview

RTI® Connext® DDS is a connectivity framework for distributed real-time applications. Connext DDS simplifies application development, deployment and maintenance and provides fast, predictable distribution of time-critical data over a variety of transport networks.

Connext DDS solutions provide a flexible data distribution infrastructure for integrating data sources of all types. At its core is the world's leading ultra-high performance, distributed networking Connext Databus. It connects data within applications as well as across devices, systems and networks. Connext DDS also delivers large data sets with microsecond performance and granular quality-of-service control. Connext DDS is a standards-based, open architecture that connects devices from deeply embedded real-time platforms to enterprise servers across a variety of networks.

With Connext DDS, you can:

- Perform complex one-to-many and many-to-many network communications.
- Customize application operation to meet various real-time, reliability, and quality-of-service goals.
- Provide application-transparent fault tolerance and application robustness.
- Use a variety of transports.

This section introduces basic concepts of middleware and common communication models, and describes how Connext DDS’s feature-set addresses the needs of real-time systems.

1.1 What is Connext DDS?

Connext DDS is a software connectivity framework for real-time distributed applications. It provides the communications service programmers need to distribute time-critical data between embedded and/or enterprise devices or nodes. Connext DDS uses the publish-subscribe communications model to make data distribution efficient and robust.
Connext DDS implements the Data-Centric Publish-Subscribe (DCPS) API within the OMG’s Data Distribution Service (DDS) for Real-Time Systems. DDS is the first standard developed for the needs of real-time systems. DCPS provides an efficient way to transfer data in a distributed system.

With Connext DDS, systems designers and programmers start with a fault-tolerant and flexible communications infrastructure that will work over a wide variety of computer hardware, operating systems, languages, and networking transport protocols. Connext DDS is highly configurable so programmers can adapt it to meet the application’s specific communication requirements.

### 1.2 Network Communications Models

The communications model underlying the network middleware is the most important factor in how applications communicate. The communications model impacts the performance, the ease to accomplish different communication transactions, the nature of detecting errors, and the robustness to different error conditions. Unfortunately, there is no “one size fits all” approach to distributed applications. Different communications models are better suited to handle different classes of application domains.

This section describes three main types of network communications models:

- Point-to-point
- Client-server
- Publish-subscribe

**Point-to-point model:**

Point-to-point is the simplest form of communication, as illustrated in Figure 1.1: Point-to-Point on the next page. The telephone is an example of an everyday point-to-point communications device. To use a telephone, you must know the address (phone number) of the other party. Once a connection is established, you can have a reasonably high-bandwidth conversation. However, the telephone does not work as well if you have to talk to many people at the same time. The telephone is essentially one-to-one communication.

TCP is a point-to-point network protocol designed in the 1970s. While it provides reliable, high-bandwidth communication, TCP is cumbersome for systems with many communicating nodes.
1.2 Network Communications Models

Figure 1.1: Point-to-Point

![Point-to-Point diagram](image)

*Point-to-point is one-to-one communication.*

**Client-server model:**

To address the scalability issues of the Point-to-Point model, developers turned to the Client-Server model. Client-server networks designate one special server node that connects simultaneously to many client nodes, as illustrated in Figure 1.2: Client-Server below.

Figure 1.2: Client-Server

![Client-Server diagram](image)

*Client-server is many-to-one communications.*

Client-server is a "many-to-one" architecture. Ordering pizza over the phone is an example of client-server communication. Clients must know the phone number of the pizza parlor to place an order. The parlor can handle many orders without knowing ahead of time where people (clients) are located. After the order
In the client-server model, each response is tied to a prior request. As a result, the response can be tailored to each request. In other words, each client makes a request (order) and each reply (pizza) is made for one specific client in mind.

The client-server network architecture works best when information is centralized, such as in databases, transaction processing systems, and file servers. However, if information is being generated at multiple nodes, a client-server architecture requires that all information are sent to the server for later redistribution to the clients. This approach is inefficient and precludes deterministic communications, since the client does not know when new information is available. The time between when the information is available on the server, and when the client asks and receives it adds a variable latency to the system.

**Publish-subscribe model:** In the publish-subscribe communications model (Figure 1.3: Publish-Subscribe on the next page), computer applications (nodes) “subscribe” to data they need and “publish” data they want to share. Messages pass directly between the publisher and the subscribers, rather than moving into and out of a centralized server. Most time-sensitive information intended to reach many people is sent by a publish-subscribe system. Examples of publish-subscribe systems in everyday life include television, magazines, and newspapers.

Publish-subscribe communication architectures are good for distributing large quantities of time-sensitive information efficiently, even in the presence of unreliable delivery mechanisms. This direct and simultaneous communication among a variety of nodes makes publish-subscribe network architecture the best choice for systems with complex time-critical data flows.

While the publish-subscribe model provides system architects with many advantages, it may not be the best choice for all types of communications, including:

- File-based transfers (alternate solution: FTP)
- Remote Method Invocation (alternate solutions: CORBA, COM, SOAP)
- Connection-based architectures (alternate solution: TCP/IP)
- Synchronous transfers (alternate solution: CORBA)
1.3 What is Middleware?

*Middleware* is a software layer between an application and the operating system. *Network middleware* isolates the application from the details of the underlying computer architecture, operating system and network stack (see Figure 1.4: Network Middleware on the next page). Network middleware simplifies the development of distributed systems by allowing applications to send and receive information without having to program using lower-level protocols such as sockets and TCP or UDP/IP.
Figure 1.4: Network Middleware

Middleware insulates applications from the raw operating-system network stack.

**Publish-subscribe communications model:** Connext DDS is based on a publish-subscribe communications model. Publish-subscribe (PS) provides a simple and intuitive way to distribute data. It decouples the software that creates and sends data—the data publishers—from the software that receives and uses the data—the data subscribers. Publishers simply declare their intent to send and then publish the data. Subscribers declare their intent to receive, then the data is automatically delivered by the middleware.

Despite the simplicity of the model, PS middleware can handle complex patterns of information flow. The use of PS middleware results in simpler, more modular distributed applications. Perhaps most importantly, PS middleware can automatically handle all network chores, including connections, failures, and network changes, eliminating the need for user applications to program all those special cases. What experienced network middleware developers know is that handling special cases accounts for over 80% of the effort and code.

### 1.4 Features of Connext DDS

Connext DDS supports mechanisms that go beyond the basic publish-subscribe model. The key benefit is that applications that use Connext DDS for their communications are entirely decoupled. Very little of their design time has to be spent on how to handle their mutual interactions. In particular, the applications never need information about the other participating applications, including their existence or locations. Connext DDS automatically handles all aspects of message delivery, without requiring any intervention from the user applications, including:
1.4 Features of Connext DDS

- determining who should receive the messages,
- where recipients are located,
- what happens if messages cannot be delivered.

This is made possible by how Connext DDS allows the user to specify Quality of Service (QoS) parameters as a way to configure automatic-discovery mechanisms and specify the behavior used when sending and receiving messages. The mechanisms are configured up-front and require no further effort on the user's part. By exchanging messages in a completely anonymous manner, Connext DDS greatly simplifies distributed application design and encourages modular, well-structured programs.

Furthermore, Connext DDS includes the following features, which are designed to meet the needs of distributed real-time applications:

- **Data-centric publish-subscribe communications**: Simplifies distributed application programming and provides time-critical data flow with minimal latency.
  - Clear semantics for managing multiple sources of the same data.
  - Efficient data transfer, customizable Quality of Service, and error notification.
  - Guaranteed periodic samples, with maximum rate set by subscriptions.
  - Notification by a callback routine on data arrival to minimize latency.
  - Notification when data does not arrive by an expected deadline.
  - Ability to send the same message to multiple computers efficiently.

- **User-definable data types**: Enables you to tailor the format of the information being sent to each application.

- **Reliable messaging**: Enables subscribing applications to specify reliable delivery of samples.

- **Multiple Communication Networks**: Multiple independent communication networks (DDS domains), each using Connext DDS, can be used over the same physical network. Applications are only able to participate in the DDS domains to which they belong. Individual applications can be configured to participate in multiple DDS domains.

- **Symmetric architecture**: Makes your application robust:
  - No central server or privileged nodes, so the system is robust to node failures.
  - Subscriptions and publications can be dynamically added and removed from the system at any time.

- **Pluggable Transports Framework**: Includes the ability to define new transport plug-ins and run over them. Connext DDS comes with a standard UDP/IP pluggable transport and a shared memory transport. It can be configured to operate over a variety of transport mechanisms, including backplanes, switched fabrics, and new networking technologies.
1.4 Features of Connext DDS

- **Multiple Built-in Transports:** Includes UDP/IP and shared memory transports.
- **Multi-language support:** Includes APIs for the C, C++ (Traditional and Modern APIs), C++/CLI, C#, and Java™ programming languages.
- **Multi-platform support:** Includes support for flavors of UNIX®, real-time operating systems, and Windows®. (Consult the RTI Connext DDS Core Libraries Platform Notes to see which platforms are supported in this release.)
- **Compliance with Standards:**
  - API complies with the DCPS layer of the OMG’s DDS specification.
  - Data types comply with OMG Interface Definition Language™ (IDL).
  - Data packet format complies with the International Engineering Consortium’s (IEC’s) publicly available specification for the RTPS wire protocol.
Chapter 2 Data-Centric Publish-Subscribe Communications

This section describes the formal communications model used by Connext DDS: the Data-Centric Publish-Subscribe (DCPS) standard. DCPS is a formalization (through a standardized API) and extension of the publish-subscribe communications model presented in 1.2 Network Communications Models on page 7.

2.1 What is DCPS?

DCPS is the portion of the OMG DDS (Data Distribution Service) Standard that addresses data-centric publish-subscribe communications. The DDS standard defines a language-independent model of publish-subscribe communications that has standardized mappings into various implementation languages. Connext DDS offers C, Traditional C++, Modern C++, C++/CLI, C#, and Java versions of the DCPS API.

The publish-subscribe approach to distributed communications is a generic mechanism that can be employed by many different types of applications. The DCPS model described in this chapter extends the publish-subscribe model to address the specific needs of real-time, data-critical applications. As you’ll see, it provides several mechanisms that allow application developers to control how communications works and how the middleware handles resource limitations and error conditions.

The “data-centric” portion of the term DCPS describes the fundamental concept supported by the design of the API. In data-centric communications, the focus is on the distribution of data between communicating applications. A data-centric system is comprised of data publishers and data subscribers. The communications are based on passing data of known types in named streams from publishers to subscribers.

In contrast, in object-centric communications the fundamental concept is the interface between the applications. An interface is comprised of a set of methods of known types (number and types of method arguments). An object-centric system is comprised of interface servers and interface clients,
and communications are based on clients invoking methods on named interfaces that are serviced by the corresponding server.

Data and object-centric communications are complementary paradigms in a distributed system. Applications may require both. However, real-time communications often fit a data-centric model more naturally.

### 2.1.1 DCPS for Real-Time Requirements

DCPS, and specifically the Connext DDS implementation, is well suited for real-time applications. For instance, real-time applications often require the following features:

- **Efficiency**

  Real-time systems require efficient data collection and delivery. Only minimal delays should be introduced into the critical data-transfer path. Publish-subscribe is more efficient than client-server in both latency and bandwidth for periodic data exchange. Publish-subscribe greatly reduces the overhead required to send data over the network compared to a client-server architecture. Occasional subscription requests, at low bandwidth, replace numerous high-bandwidth client requests. Latency is also reduced, since the outgoing request message time is eliminated. As soon as a new DDS sample becomes available, it is sent to the corresponding subscriptions.

- **Determinism**

  Real-time applications often care about the determinism of delivering periodic data as well as the latency of delivering event data. Once buffers are introduced into a data stream to support reliable connections, new data may be held undelivered for a unpredictable amount of time while waiting for confirmation that old data was received. Since publish-subscribe does not inherently require reliable connections, implementations, like Connext DDS, can provide configurable trade-offs between the deterministic delivery of new data and the reliable delivery of all data.

- **Flexible delivery bandwidth**

  Typical real-time systems include both real-time and non-real-time nodes. The bandwidth requirements for these nodes—even for the same data—are quite different. For example, an application may be sending DDS samples faster than a non-real-time application is capable of handling. However, a real-time application may want the same data as fast as it is produced. DCPS allows subscribers to the same data to set individual limits on how fast data should be delivered to each subscriber. This is similar to how some people get a newspaper every day while others can subscribe to only the Sunday paper.

- **Thread awareness**
Real-time communications must work without slowing the thread that sends DDS samples. On the receiving side, some data streams should have higher priority so that new data for those streams are processed before lower priority streams.

*Connext DDS* provides user-level configuration of its internal threads that process incoming data. Users may configure *Connext DDS* so that different threads are created with different priorities to process received data of different data streams.

- **Fault-tolerant operation**

  Real-time applications are often in control of systems that are required to run in the presence of component failures. Often, those systems are safety critical or carry financial penalties for loss of service. The applications running those systems are usually designed to be fault-tolerant using redundant hardware and software. Backup applications are often “hot” and interconnected to primary systems so that they can take over as soon as a failure is detected.

  Publish-subscribe is capable of supporting many-to-many connectivity with redundant *DataWriters* and *DataReaders*. This feature is ideal for constructing fault-tolerant or high-availability applications with redundant nodes and robust fault detection and handling services.

- DCPS, and thus *Connext DDS*, was designed and implemented specifically to address the requirements above through configuration parameters known as QosPolicies defined by the DCPS standard (see 4.2 QosPolicies on page 176). 2.2 DDS Data Types, Topics, Keys, Instances, and Samples below introduces basic DCPS terminology and concepts.

### 2.2 DDS Data Types, Topics, Keys, Instances, and Samples

In data-centric communications, the applications participating in the communication need to share a common view of the types of data being passed around.

Within different programming languages there are several ‘primitive’ data types that all users of that language naturally share (integers, floating point numbers, characters, booleans, etc.). However, in any non-trivial software system, specialized data types are constructed out of the language primitives. So the data to be shared between applications in the communication system could be structurally simple, using the primitive language types mentioned above, or it could be more complicated, using, for example, C and C++ structs, like this:
2.3 Data Topics — What is the Data Called?

Within a set of applications using DCPS, the different applications do not automatically know the structure of the data being sent, nor do they necessarily interpret it in the same way (if, for instance, they use different operating systems, were written with different languages, or were compiled with different compilers). There must be a way to share not only the data, but also information about how the data is structured.

In DCPS, data definitions are shared among applications using OMG IDL, a language-independent means of describing data. For more information on data types and IDL, see Data Types and DDS Data Samples (Chapter 3 on page 27).

### 2.3 Data Topics — What is the Data Called?

Shared knowledge of the data types is a requirement for different applications to communicate with DCPS. The applications must also share a way to identify which data is to be shared. Data (of any data type) is uniquely distinguished by using a name called a **Topic**. By definition, a **Topic** corresponds to a single data type. However, several **Topics** may refer to the same data type.

**Topics** interconnect **DataWriters** and **DataReader**s. A **DataWriter** is an object in an application that tells Connext DDS (and indirectly, other applications) that it has some values of a certain **Topic**. A corresponding **DataReader** is an object in an application that tells Connext DDS that it wants to receive values for the same **Topic**. And the data that is passed from the **DataWriter** to the **DataReader** is of the data type associated with the **Topic**. **DataWriters** and **DataReader**s are described more in 2.5 DataWriters/Publishers and DataReaders/Subscribers on page 20.

For a concrete example, consider a system that distributes stock quotes between applications. The applications could use a data type called StockPrice. There could be multiple **Topics** of the StockPrice data type, one for each company’s stock, such as IBM, MSFT, GE, etc. Each **Topic** uses the same data type.

**Data Type: StockPrice**

```c
struct StockPrice {
    float price;
    Time timeTimestamp;
};
```

**Topic: “IBM”**

**Topic: “MSFT”**
Now, an application that keeps track of the current value of a client’s portfolio would subscribe to all of the topics of the stocks owned by the client. As the value of each stock changes, the new price for the corresponding topic is published and sent to the application.

2.4 DDS Samples, Instances, and Keys

The value of data associated with a *Topic* can change over time. The different values of the *Topic* passed between applications are called DDS samples. In our stock-price example, DDS samples show the price of a stock at a certain point in time. So each DDS sample may show a different price.

For a data type, you can select one or more fields within the data type to form a *key*. A *key* is something that can be used to uniquely identify one *instance* of a *Topic* from another *instance* of the same *Topic*. Think of a key as a way to sub-categorize or group related data values for the same *Topic*. Note that not all data types are defined to have keys, and thus, not all topics have keys. For topics without keys, it's as if there is only a single instance of that topic.

However, for *Topics* with keys, a unique value for the key identifies a unique *instance* of the *Topic*. DDS samples are then updates to particular instances of a *Topic*.

For example, let’s change the *StockPrice* data type to include the symbol of the stock. Then instead of having a *Topic* for every stock, which would result in hundreds or thousands of *Topics* and related *DataWriters* and *DataReaders*, each application would only have to publish or subscribe to a single *Topic*, say “StockPrices.” Successive values of a stock would be presented as successive DDS samples of an instance of “StockPrices”, with each instance corresponding to a single stock symbol.

**Data Type: StockPrice**

```c
struct StockPrice {
    float price;
    Time time;
    @key char *symbol;
};
```

Instance 1 = (Topic: “StockPrices”) + (Key: “MSFT”)

    sample a, price = $28.00
    sample b, price = $27.88

Instance 2 = (Topic: “StockPrices”) + (Key: “IBM”)

    sample a, price = $74.02
    sample b, price = $73.50

Etc.
Applications can subscribe to a *Topic* and receive DDS samples for many different instances. Applications can publish DDS samples of one, all, or any number of instances of a *Topic*. Many quality of service parameters actually apply on a *per instance* basis. Keys are also useful for subscribing to a group of related data streams (instances) without pre-knowledge of which data streams (instances) exist at runtime.

For example, just by subscribing to “StockPrices,” an application can get values for all of the stocks through a single topic. In addition, the application does not have to subscribe explicitly to any particular stock, so that if a new stock is added, the application will immediately start receiving values for that stock as well.

Many quality of service (QoS) parameters apply on a per-instance basis because each instance is a unique object and therefore has its own lifecycle, owner, and resource limits.

To summarize, the unique values of data being passed using DCPS are called DDS samples. A DDS sample is a combination of a *Topic*, an *instance*, and the actual user data of a certain data type. As seen in Figure 2.1: Relationship of Topics, Keys, and Instances below, a *Topic* identifies data of a single type, ranging from one single instance to a whole collection of instances of that given topic for keyed data types. For more information, see Data Types and DDS Data Samples (Chapter 3 on page 27) and Working with Topics (Chapter 5 on page 215).

**Figure 2.1: Relationship of Topics, Keys, and Instances**

![Diagram of Topics, Keys, and Instances]

By using keys, a *Topic* can identify a collection of data-object instances.

See Chapter 6 Working with Instances on page 257.
2.5 DataWriters/Publishers and DataReaders/Subscribers

In DCPS, applications must use APIs to create entities (objects) in order to establish publish-subscribe communications between each other. The entities and terminology associated with the data itself have been discussed already—Topics, keys, instances, DDS samples. This section will introduce the DCPS entities that user code must create to send and receive the data. Note that Entity is actually a basic DCPS concept. In object-oriented terms, Entity is the base class from which other DCPS classes—Topic, DataWriter, DataReader, Publisher, Subscriber, DomainParticipants—derive. For general information on Entities, see DDS Entities (Chapter 4 on page 165).

The sending side uses objects called Publishers and DataWriters. The receiving side uses objects called Subscribers and DataReaders. Figure 2.2: Overview below illustrates the relationship of these objects.

Figure 2.2: Overview

- An application uses DataWriters to send data. A DataWriter is associated with a single Topic. You can have multiple DataWriters and Topics in a single application. In addition, you can have more than one DataWriter for a particular Topic in a single application.
- A Publisher is the DCPS object responsible for the actual sending of data. Publishers own and manage DataWriters. A DataWriter can only be owned by a single Publisher while a Publisher can own many DataWriters. Thus the same Publisher may be sending data for many different Topics of
2.5 DataWriters/Publishers and DataReaders/Subscribers

different data types. When user code calls the `write()` method on a `DataWriter`, the DDS data sample is passed to the `Publisher` object which does the actual dissemination of data on the network. For more information, see Sending Data (Chapter 7 on page 270).

- The association between a `DataWriter` and a Publisher is often referred to as a publication although you never create a DCPS object known as a publication.

- An application uses `DataReader`s to access data received over DCPS. A `DataReader` is associated with a single `Topic`. You can have multiple `DataReader`s and `Topics` in a single application. In addition, you can have more than one `DataReader` for a particular `Topic` in a single application.

- A `Subscriber` is the DCPS object responsible for the actual receipt of published data. `Subscribers` own and manage `DataReader`s. A `DataReader` can only be owned by a single `Subscriber` while a `Subscriber` can own many `DataReader`s. Thus the same `Subscriber` may receive data for many different `Topics` of different data types. When data is sent to an application, it is first processed by a `Subscriber`; the DDS data sample is then stored in the appropriate `DataReader`. User code can either register a `listener` to be called when new data arrives or actively poll the `DataReader` for new data using its `read()` and `take()` methods. For more information, see Receiving Data (Chapter 8 on page 484).

- The association between a `DataReader` and a Subscriber is often referred to as a subscription although you never create a DCPS object known as a subscription.

**Example:**

The publish-subscribe communications model is analogous to that of magazine publications and subscriptions. Think of a publication as a weekly periodical such as *Newsweek®*. The `Topic` is the name of the periodical (in this case the string "Newsweek"). The `type` specifies the format of the information, e.g., a printed magazine. The `user data` is the contents (text and graphics) of each DDS sample (weekly issue). The middleware is the distribution service (usually the US Postal service) that delivers the magazine from where it is created (a printing house) to the individual subscribers (people’s homes). This analogy is illustrated in Figure 2.3: An Example of Publish-Subscribe on the next page. Note that by subscribing to a publication, subscribers are requesting current and future DDS samples of that publication (such as once a week in the case of *Newsweek*), so that as new DDS samples are published, they are delivered without having to submit another request for data.
The publish-subscribe model is analogous to publishing magazines. The Publisher sends DDS samples of a particular Topic to all Subscribers of that Topic. With Newsweek® magazine, the Topic would be "Newsweek." The DDS sample consists of the data (articles and pictures) sent to all Subscribers every week. The middleware (Connext DDS) is the distribution channel: all of the planes, trucks, and people who distribute the weekly issues to the Subscribers.

By default, each DDS sample is propagated individually, independently, and uncorrelated with other DDS samples. However, an application may request that several DDS samples be sent as a coherent set, so that they may be interpreted as such on the receiving side.

### 2.6 DDS Domains and DomainParticipants

You may have several independent DCPS applications all running on the same set of computers. You may want to isolate one (or more) of those applications so that it isn’t affected by the others. To address this issue, DCPS has a concept called DDS domains.

DDS domains represent logical, isolated, communication networks. Multiple applications running on the same set of hosts on different DDS domains are completely isolated from each other (even if they are on the same machine). DataWriters and DataReaders belonging to different DDS domains will never exchange data.

Applications that want to exchange data using DCPS must belong to the same DDS domain. To belong to a DDS domain, DCPS APIs are used to configure and create a DomainParticipant with a specific Domain Index. DDS domains are differentiated by the domain index (an integer value). Applications that have created DomainParticipants with the same domain index belong to the same DDS domain. DomainParticipants own Topics, Publishers, and Subscribers, which in turn owns DataWriters and DataReaders. Thus all DCPS Entities belong to a specific DDS domain.

An application may belong to multiple DDS domains simultaneously by creating multiple DomainParticipants with different domain indices. However, Publishers/DataWriters and Subscribers/DataReaders only belong to the DDS domain in which they were created.
As mentioned before, multiple DDS domains may be used for application isolation, which is useful when you are testing applications using computers on the same network or even the same computers. By assigning each user different domains, one can guarantee that the data produced by one user’s application won’t accidentally be received by another. In addition, DDS domains may be a way to scale and construct larger systems that are composed of multi-node subsystems. Each subsystem would use an internal DDS domain for intra-system communications and an external DDS domain to connect to other subsystems.

For more information, see Working with DDS Domains (Chapter 9 on page 600).

2.7 Quality of Service (QoS)

The publish-subscribe approach to distributed communications is a generic mechanism that can be employed by many different types of systems. The DCPS model described here extends the publish-subscribe model to address the needs of real-time, data-critical applications. It provides standardized mechanisms, known as Quality of Service Policies, that allow application developers to configure how communications occur, to limit resources used by the middleware, to detect system incompatibilities and setup error handling routines.

2.7.1 Controlling Behavior with Quality of Service (QoS) Policies

QosPolicies control many aspects of how and when data is distributed between applications. The overall QoS of the DCPS system is made up of the individual QosPolicies for each DCPS Entity. There are QosPolicies for Topics, DataWriters, Publishers, DataReaders, Subscribers, and DomainParticipants.

On the publishing side, the QoS of each Topic, the Topic’s DataWriter, and the DataWriter’s Publisher all play a part in controlling how and when DDS samples are sent to the middleware. Similarly, the QoS of the Topic, the Topic’s DataReader, and the DataReader’s Subscriber control behavior on the subscribing side.

Users will employ QosPolicies to control a variety of behaviors. For example, the DEADLINE policy sets up expectations of how often a DataReader expects to see DDS samples. The OWNERSHIP and OWNERSHIP_STRENGTH policy are used together to configure and arbitrate whose data is passed to the DataReader when there are multiple DataWriters for the same instance of a Topic. The HISTORY policy specifies whether a DataWriter should save old data to send to new subscriptions that join the network later. Many other policies exist and they are presented in 4.2 QosPolicies on page 176.

Some QosPolicies represent “contracts” between publications and subscriptions. For communications to take place properly, the QosPolicies set on the DataWriter side must be compatible with corresponding policies set on the DataReader side.

For example, the RELIABILITY policy is set by the DataWriter to state whether it is configured to send data reliably to DataReaders. Because it takes additional resources to send data reliably, some DataWriters may only support a best-effort level of reliability. This implies that for those DataWriters, Connext DDS will not spend additional effort to make sure that the data sent is received by DataReaders or resend any lost data. However, for certain applications, it could be imperative that their DataReaders receive every
piece of data with total reliability. Running a system where the DataWriters have not been configured to support the DataReaders could lead to erratic failures.

To address this issue, and yet keep the publications and subscriptions as decoupled as possible, DCPS provides a way to detect and notify when QosPolicies set by DataWriters and DataReaders are incompati-

ble. DCPS employs a pattern known as RxO (Requested versus Offered). The DataReader sets a “requested” value for a particular QosPolicy. The DataWriter sets an “offered” value for that QosPolicy. When Connext DDS matches a DataReader to a DataWriter, QosPolicies are checked to make sure that all requested values can be supported by the offered values.

Note that not all QosPolicies are constrained by the RxO pattern. For example, it does not make sense to compare policies that affect only the DataWriter but not the DataReader or vice versa.

If the DataWriter cannot satisfy the requested QosPolicies of a DataReader, Connext DDS will not connect the two DDS entities and will notify the applications on each side of the incompatibility if so con-

figured.

For example, a DataReader sets its DEADLINE QoS to 4 seconds—that is, the DataReader is requesting that it receive new data at least every 4 seconds.

In one application, the DataWriter sets its DEADLINE QoS to 2 seconds—that is, the DataWriter is committing to sending data at least every 2 seconds. This writer can satisfy the request of the reader, and thus, Connext DDS will pass the data sent from the writer to the reader.

In another application, the DataWriter sets its DEADLINE QoS to 5 seconds. It only commits to sending data at 5 second intervals. This will not satisfy the request of the DataReader. Connext DDS will flag this incompatibility by calling user-installed listeners in both DataWriter and DataReader applications and not pass data from the writer to the reader.

For a summary of the QosPolicies supported by Connext DDS, see 4.2 QosPolicies on page 176.

2.8 Application Discovery

The DCPS model provides anonymous, transparent, many-to-many communications. Each time an application sends a DDS sample of a particular Topic, the middleware distributes the DDS sample to all the applications that want that Topic. The publishing application does not need to specify how many applications receive the Topic, nor where those applications are located. Similarly, subscribing applications do not specify the location of the publications. In addition, new publications and subscriptions of the Topic can appear at any time, and the middleware will automatically interconnect them.

So how is this all done? Ultimately, in each application for each publication, Connext DDS must keep a list of applications that have subscribed to the same Topic, nodes on which they are located, and some additional QoS parameters that control how the data is sent. Also, Connext DDS must keep a list of applications and publications for each of the Topics to which the application has subscribed.
Propagation of this information (the existence of publications and subscriptions and associated QoS) between applications by Connext DDS is known as the discovery process. While the DDS (DCPS) standard does not specify how discovery occurs, Connext DDS uses a standard protocol RTPS for both discovery and formatting on-the-wire packets.

When a DomainParticipant is created, Connext DDS sends out packets on the network to announce its existence. When an application finds out that another application belongs to the same DDS domain, then it will exchange information about its existing publications and subscriptions and associated QoS with the other application. As new DataWriters and DataReaders are created, this information is sent to known applications.

The Discovery process is entirely configurable by the user and is discussed extensively in Discovery (Chapter 15 on page 764).
Part 2: Core Concepts

This section includes:

- Data Types and DDS Data Samples (Chapter 3 on page 27)
- DDS Entities (Chapter 4 on page 165)
- Working with Topics (Chapter 5 on page 215)
- Working with Instances (Chapter 6 on page 257)
- Sending Data (Chapter 7 on page 270)
- Receiving Data (Chapter 8 on page 484)
- Working with DDS Domains (Chapter 9 on page 600)
- Building Applications (Chapter 10 on page 680)
Chapter 3 Data Types and DDS Data Samples

Note: Information in this chapter is complemented by information in the RTI Connext DDS Core Libraries Extensible Types Guide.

How data is stored or laid out in memory can vary from language to language, compiler to compiler, operating system to operating system, and processor to processor. This combination of language/compiler/operating system/processor is called a platform. Any modern middleware must be able to take data from one specific platform (say C/gcc 7.3/Linux/Arm v8) and transparently deliver it to another (for example, Java/JDK 11/Windows/Pentium). This process is commonly called serialization/deserialization, or marshalling/demarshalling.

Messaging products have typically taken one of two approaches to this problem:

1. **Do nothing.** Messages consist only of opaque streams of bytes. The JMS BytesMessage is an example of this approach.

2. **Send everything, every time.** Self-describing messages are at the opposite extreme, embedding full reflective information, including data types and field names, with each message. The JMS MapMessage and the messages in TIBCO Rendezvous are examples of this approach.

The “do nothing” approach is lightweight on its surface but forces you, the user of the middleware API, to consider all data encoding, alignment, and padding issues. The “send everything” alternative results in large amounts of redundant information being sent with every packet, impacting performance.

Connext DDS takes an intermediate approach. Just as objects in your application program belong to some data type, DDS data samples sent on the same Connext DDS topic share a data type. This type defines the fields that exist in the DDS data samples and what their constituent types are. The middleware stores and propagates this meta-information separately from the individual DDS data
samples, allowing it to propagate DDS samples efficiently while handling byte ordering and alignment issues for you.

To publish and/or subscribe to data with Connext DDS, you will carry out the following steps:

1. Select a type to describe your data.

   You have a number of choices. You can choose one of these options, or you can mix and match them.

   - Use a built-in type provided by the middleware.

     This option may be sufficient if your data typing needs are very simple. If your data is highly structured, or you need to be able to examine fields within that data for filtering or other purposes, this option may not be appropriate. The built-in types are described in 3.2 Built-in Data Types on page 38.

   - Use the RTI Code Generator to define a type at compile-time using a language-independent description language.

     Code generation offers two strong benefits not available with dynamic type definition: (1) it allows you to share type definitions across programming languages, and (2) because the structure of the type is known at compile time, it provides rigorous static type safety.

     The RTI Code Generator accepts input in the following formats:

     - **OMG IDL.** This format is a standardized component of the DDS specification. It describes data types with a C++-like syntax. A link to the latest specification can be found here: [https://www.omg.org/spec/IDL](https://www.omg.org/spec/IDL). This format is described in 3.3 Creating User Data Types with IDL on page 75.

     - **XML in a DDS-specific format.** This XML format is terser, and therefore easier to read and write by hand, than an XSD file. It offers the general benefits of XML-extensibility and ease of integration, while fully supporting DDS-specific data types and concepts. A link to the latest specification, including a description of the XML format, can be found here: [https://www.omg.org/spec/DDS-XTypes/](https://www.omg.org/spec/DDS-XTypes/). This format is described in 3.4 Creating User Data Types with Extensible Markup Language (XML) on page 121.

     - **XSD format.** You can describe data types with XML schemas (XSD). A link to the latest specification, including a description of the XSD format, can be found here: [https://www.omg.org/spec/DDS-XTypes/](https://www.omg.org/spec/DDS-XTypes/). This format is described in 3.5 Creating User Data Types with XML Schemas (XSD) on page 131.

   - Define a type programmatically at run time.

     This method may be appropriate for applications with dynamic data description needs: applications for which types change frequently or cannot be known ahead of time. It is described in 3.8.2 Defining New Types on page 154.
2. Register your type with a logical name.

If you've chosen to use a built-in type instead of defining your own, you can omit this step; the middleware pre-registers the built-in types for you.

This step is described in the 3.8.2 Defining New Types on page 154.

3. Create a Topic using the type name you previously registered.

If you've chosen to use a built-in type instead of defining your own, you will use the API constant corresponding to that type's name.

Creating and working with Topics is discussed in Working with Topics (Chapter 5 on page 215).

4. Create one or more DataWriters to publish your data and one or more DataReaders to subscribe to it.

The concrete types of these objects depend on the concrete data type you've selected, in order to provide you with a measure of type safety.

Creating and working with DataWriters and DataReaders are described in Sending Data (Chapter 7 on page 270) and Receiving Data (Chapter 8 on page 484), respectively.

Whether publishing or subscribing to data, you will need to know how to create and delete DDS data samples and how to get and set their fields. These tasks are described in 3.9 Working with DDS Data Samples on page 158.

3.1 Introduction to the Type System

A user data type is any custom type that your application defines for use with Connext DDS. It may be a structure, a union, a value type, an enumeration, or a typedef (or language equivalents).

Your application can have any number of user data types. They can be composed of any of the primitive data types listed below or of other user data types.

Only structures, unions, and value types may be read and written directly by Connext DDS; enums, typedefs, and primitive types must be contained within a structure, union, or value type. In order for a DataReader and DataWriter to communicate with each other, the data types associated with their respective Topic definitions must be consistent according to the Type-Consistency Enforcement rules configured using the TypeConsistencyEnforcementQosPolicy on the DataReaderQos (see 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594).

- octet, char, wchar
- int16 or short, uint16 or unsigned short
- int32 or long, uint32 or unsigned long
- int64 or long long, uint64 or unsigned long long
3.1.1 Sequences

- float
- double, long double
- boolean
- enum (with or without explicit values)
- bounded and unbounded string and wstring

The following type-building constructs are also supported:

- module (also called a package or namespace)
- pointer
- array of primitive or user type elements
- bounded/unbounded sequence of elements\(^1\)—a sequence is a variable-length ordered collection, such as a vector or list
- typedef
- union
- struct, a complex type that supports inheritance and other object-oriented features
- value type, a deprecated type that is treated identically to a struct for backward compatibility with existing type definitions

To use a data type with Connext DDS, you must define that type in a way the middleware understands and then register the type with the middleware. These steps allow Connext DDS to serialize, deserialize, and otherwise operate on specific types. They will be described in detail in the following sections.

### 3.1.1 Sequences

A sequence contains an ordered collection of elements that are all of the same type. The operations supported in the sequence are documented in the API Reference HTML documentation, which is available for all supported programming languages (select Modules, RTI Connext DDS API Reference, Infrastructure Module, Sequence Support).

Java sequences implement the java.util.List interface from the standard Collections framework.

In the Modern C++ API, a sequence of type T maps to the type \texttt{std::vector<T>}, or to a type with a similar interface, depending on the options and whether it is bounded or unbounded. See 3.3.4 Translations for IDL Types on page 79.

\(^1\)Sequences of sequences are not supported directly. To work around this constraint, typedef the inner sequence and form a sequence of that new type.
Elements in a sequence are accessed with their index, just like elements in an array. Indices start at zero in all APIs except Ada. In Ada, indices start at 1. Unlike arrays, however, sequences can grow in size. A sequence has two sizes associated with it: a physical size (the "maximum") and a logical size (the "length"). The physical size indicates how many elements are currently allocated by the sequence to hold; the logical size indicates how many valid elements the sequence actually holds. The length can vary from zero up to the maximum. Elements cannot be accessed at indices beyond the current length.

A sequence may be declared as bounded or unbounded. A sequence's "bound" is the maximum number of elements that the sequence can contain at any one time. A finite bound is very important because it allows Connext DDS to preallocate buffers to hold serialized and deserialized samples of your types; these buffers are used when communicating with other nodes in your distributed system. If a sequence has no bound, Connext DDS will not know how large to allocate its buffers and will therefore have to allocate them on the fly as individual samples are read and written—impacting the latency and determinism of your application.

By default, any unbounded sequences found in an IDL file will be given a default bound of 100 elements. This default value can be overwritten using the RTI Code Generator’s -sequenceSize command-line argument (see the RTI Code Generator User's Manual).

When using the C, C++, Java, or .NET APIs, you can change the default behavior and use truly unbounded sequences by using RTI Code Generator’s -unboundedSupport command-line argument. When using this option, the generated code will deserialize incoming samples as follows:

- First, it will release previous memory associated with the unbounded sequences. The memory associated with an unbounded member is not released until the sample containing the member is reused.
- Second, it will allocate new memory to accommodate the actual size of the unbounded sequences.

To configure unbounded support for code generated with rtiddsgen -unboundedSupport or for DynamicDataWriters/DynamicDataReaders for Topics of types that contain unbounded sequences:

1. Use these threshold QoS properties:
   - `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size` on the `DataWriter`
   - `dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size` on the `DataReader`

2. Set the QoS value `reader_resource_limits.dynamically_allocate_fragmented_samples` on the `DataReader` to true.

3. For the Java API, also set these properties accordingly for the Java serialization buffer:
   - `dds.data_writer.history.memory_manager.java_stream.min_size`
   - `dds.data_writer.history.memory_manager.java_stream.trim_to_size`
3.1.2 Strings and Wide Strings

- dds.data_reader.history.memory_manager.java_stream.min_size
- dds.data_reader.history.memory_manager.java_stream.trim_to_size

See also:
- 3.2.7.2 Unbounded Built-in Types on page 73
- 22.1.3 Writer-Side Memory Management when Using Java on page 931
- 22.2.2 Reader-Side Memory Management when Using Java on page 935
- 3.10 Data Sample Serialization Limits on page 164

3.1.2 Strings and Wide Strings

Connext DDS supports both strings consisting of single-byte characters (the IDL string type) and strings consisting of wide characters (IDL wstring). The wide characters supported by Connext DDS are large enough to store two-byte Unicode/UTF16 characters.

Like sequences, strings may be bounded or unbounded. A string's "bound" is its maximum length (not counting the trailing NULL character in C and C++).

In the Modern C++ API strings map to std::string or to a type with a similar interface, depending on the options. See Table 3.8 Specifying Data Types in IDL for the Modern C++ API in 3.3.4 Translations for IDL Types on page 79.

In C and Traditional C++, strings are mapped to char*. Optionally, the mapping in Traditional C++ can be changed to std::string by generating code with the option -useStdString.

By default, any unbounded string found in an IDL file will be given a default bound of 255 elements. This default value can be overwritten using the RTI Code Generator's -stringSize command-line argument (see the RTI Code Generator User's Manual).

When using the C, C++, Java, or .NET APIs, you can change the default behavior and use truly unbounded strings by using Code Generator's -unboundedSupport command-line argument. When using this option, the generated code will deserialize incoming samples as follows:

- First, it will release previous memory associated with the unbounded strings. The memory associated with an unbounded member is not released until the sample containing the member is reused.
- Second, it will allocate new memory to accommodate the actual size of the unbounded strings.
3.1.2 Strings and Wide Strings

To configure unbounded support for code generated with rtiddsgen -unboundedSupport or for DynamicDataWriters/DynamicDataReaders for Topics of types that contain unbounded strings or wide strings:

1. Use these threshold QoS properties:
   - `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size` on the `DataWriter`
   - `dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size` on the `DataReader`

2. Set the QoS value `reader_resource_limits.dynamically_allocate_fragmented_samples` on the `DataReader` to true.

3. For the Java API, also set these properties accordingly for the Java serialization buffer:
   - `dds.data_writer.history.memory_manager.java_stream.min_size`
   - `dds.data_writer.history.memory_manager.java_stream.trim_to_size`
   - `dds.data_reader.history.memory_manager.java_stream.min_size`
   - `dds.data_reader.history.memory_manager.java_stream.trim_to_size`

See also:

- 3.2.7.2 Unbounded Built-in Types on page 73
- 22.1.3 Writer-Side Memory Management when Using Java on page 931
- 22.2.2 Reader-Side Memory Management when Using Java on page 935
- 3.10 Data Sample Serialization Limits on page 164

### 3.1.2.1 IDL String Encoding

The “Extensible and Dynamic Topic Types for DDS specification” ([https://www.omg.org/spec/DDS-XTypes/](https://www.omg.org/spec/DDS-XTypes/)) standardizes the default encoding for strings to UTF-8. This encoding shall be used as the wire format. Language bindings may use the representation that is most natural in that particular language. If this representation is different than UTF-8, the language binding shall manage the transformation to/from the UTF-8 wire representation.

For example, in Java, IDL strings are mapped to Java `String`, which represents a string in the UTF-16 format. *Connext DDS* handles the conversion to/from UTF-8 when serializing/deserializing strings in Java.

As an extension, *Connext DDS* offers ISO_8859_1 as an alternative string wire encoding.

This section describes the encoding for IDL strings across different languages in *Connext DDS* and how to configure that encoding.
3.1.2 Strings and Wide Strings

- C, Traditional C++

IDL strings are mapped to a NULL-terminated array of DDS_Char (char*). Users are responsible for using the right character encoding (UTF-8 or ISO_8859_1) when populating the string values. This applies to all generated code, DynamicData, and Built-in data types. The middleware does not transform from the language binding encoding to the wire encoding.

- Modern C++

IDL strings are mapped to std::string, std::string, which contains any sequence of bytes. Users are responsible for using the right character encoding (UTF-8 or ISO_8859_1) when populating the string values. The middleware does not transform from the language binding encoding to the wire encoding. This applies to all generated code, DynamicData, and Built-in types.

- Ada

IDL strings are mapped to DDS.String, which is equivalent to a NULL-terminated array of DDS_Char (char*). Users are responsible for using the right character encoding (UTF-8 or ISO_8859_1) when populating the string values. The middleware does not transform from the language binding encoding to the wire encoding. This applies to all generated code and Built-in types.

- Java

IDL strings are mapped to Java String, which represents a string in the UTF-16 format. Connext DDS handles the conversion to/from UTF-8/ISO_8859_1 when serializing/deserializing strings. For generated code and Built-in data types, you can configure the IDL wire string encoding on a per-endpoint basis using the following properties:

- dds.data_reader.type_support.cdr_string_encoding_kind
- dds.data_writer.type_support.cdr_string_encoding_kind

These properties can be set at the endpoint level or the participant level. The only values currently supported are UTF-8 and ISO-8859-1. By default, the wire character encoding is assumed to be UTF-8.

For DynamicData, the user can configure the IDL wire string encoding by setting the value of string_character_encoding in DynamicDataProperty_t. The following values are supported:

- StandardCharsets.ISO_8859_1
- StandardCharsets.UTF_8 (default)

- .NET

IDL strings are mapped to .NET System::String in C++/CLI and string in C#. The conversion to/-from UTF-8/ISO_8859_1 when serializing/deserializing strings is automatically handled by Connext.
DDS. For generated code and built-in data types, you can configure the IDL wire string encoding on a per-endpoint basis using the following properties:

- `dds.data_reader.type_support.cdr_string_encoding_kind`
- `dds.data_writer.type_support.cdr_string_encoding_kind`

These properties can be set at the endpoint level or the participant level. The only values currently supported are UTF-8 and ISO-8859-1. By default, the wire character encoding is assumed to be UTF-8.

For DynamicData, you can configure the IDL wire string encoding by setting the value of `string_character_encoding` in `DynamicDataProperty_t`. The following values are supported:

- `StringEncodingKind::UTF_8` (default)
- `StringEncodingKind::ISO_8859_1`

### 3.1.2.1.1 Unicode Normalization when Using UTF-8 Encoding

Connext DDS does not normalize the content of the IDL string fields when they are serialized and sent on the wire. It is responsibility of the application to do that when needed.

Because the content of the string fields is not guaranteed to be normalized, by default, Connext DDS normalizes the UTF-8 IDL string values and the literals they are compared with in the filter expression and/or filter parameters before the filtering evaluation occurs. The normalization affects the following features:

- ContentFilteredTopics (see 5.4 ContentFilteredTopics on page 227)
- Query conditions (see 4.6.7 ReadConditions and QueryConditions on page 209)
- TopicQueries (see Chapter 24 Topic Queries on page 977)
- MultiChannel DataWriters (see Chapter 20 Multi-channel DataWriters on page 896)

You can turn off filtering normalization by using the `DomainParticipant's` Property Qos property `dds-domain_participant.filtering_unicode_normalization` (see 5.4.9 Unicode Normalization on page 247).

### 3.1.2.1.2 Filtering Character Encoding

The following filtering features use UTF-8 character encoding by default for IDL strings:

- ContentFilteredTopics (see 5.4 ContentFilteredTopics on page 227)
- Query conditions (see 4.6.7 ReadConditions and QueryConditions on page 209)
- TopicQueries (see Chapter 24 Topic Queries on page 977)
- MultiChannel DataWriters (see Chapter 20 Multi-channel DataWriters on page 896)
If the encoding of the IDL strings is ISO 8859-1, change the default filtering behavior by setting the DomainParticipant's Property Qos property `dds.domain_participant.filtering_character_encoding` to ISO-8859-1. For additional information about this property, see 5.4.8 Character Encoding on page 246.

### 3.1.2.2 IDL Wide Strings Encoding

The “Extensible and Dynamic Topic Types for DDS specification” ([https://www.omg.org/spec/DDS-XTypes/](https://www.omg.org/spec/DDS-XTypes/)) standardizes the default encoding for wide strings to UTF-16. This encoding shall be used as the wire format.

When the data representation is Extended CDR version 1, wide-string characters have a size of 4 bytes on the wire with UTF-16 encoding. When the data representation is Extended CDR version 2, wide-string characters have a size of 2 bytes on the wire with UTF-16 encoding.

Language bindings may use the representation that is most natural in that particular language. If this representation is different from UTF-16, the language binding shall manage the transformation to/from the UTF-16 wire representation.

- **C, Traditional C++**
  
  IDL wide strings are mapped to a NULL-terminated array of DDS_Wchar (DDS_Wchar*). DDS_WChar is an unsigned 2-byte integer. Users are responsible for using the right character encoding (UTF-16) when populating the wide-string values. This applies to all generated code, DynamicData, and Built-in data types. Connext DDS does not transform from the language binding encoding to the wire encoding.

- **Modern C++**
  
  IDL wide strings are mapped to `std::wstring`, which contains a sequence of wchar_t. This applies to all generated code, DynamicData, and Built-in data types. When serializing/deserializing, Connext DDS assumes that a wchar_t contains a code unit in UTF-16 encoding, even if the size of wchar_t is 4 bytes.

- **Ada**
  
  IDL wide strings are mapped to Standard.DDS.Wide_String, which is a NULL-terminated array of Standard.Wide_Character with UTF-16 encoding. This applies to all generated code and Built-in data types.

- **Java**
  
  IDL wide strings are mapped to Java String, which represents a string in the UTF-16 format. This applies to all generated code, DynamicData, and Built-in data types.

- **.NET**
IDL wide strings are mapped to `System::String` in C++/CLI and `string` in C#. These types use the UTF-16 character encoding form. This applies to all generated code, DynamicData, and Built-in data types.

### 3.1.2.2.1 Unicode Normalization when Using UTF-16 Encoding

Connext DDS does not normalize the content of the IDL wstring fields when they are serialized and sent on the wire. It is responsibility of the application to do that when needed.

Unlike with IDL strings, Connext DDS does not normalize the UTF-16 strings used by the filtering operations, either.

### 3.1.3 Introduction to TypeCode

Type schemas—the names and definitions of a type and its fields—are represented by TypeCode objects (known as DynamicType in the Modern C++ API). A type code value consists of a type code kind (see the `TCKind` enumeration below) and a list of members. For compound types like structs and arrays, this list will recursively include one or more type code values.

```cpp
enum TCKind {
    TK_NULL,
    TK_SHORT,
    TK_LONG,
    TK_USHORT,
    TK ULONG,
    TK_FLOAT,
    TK DOUBLE,
    TK_BOOLEAN,
    TK_CHAR,
    TK OCTET,
    TK STRUCT,
    TK UNION,
    TK ENUM,
    TK STRING,
    TK_SEQUENCE,
    TK ARRAY,
    TK ALIAS,
    TK LONG,
    TK ULONG,
    TK LONGLONG,
    TK ULONGLONG,
    TK LONGDOUBLE,
    TK WCHAR,
    TK WSTRING,
    TK VALUE
}
```

Type codes unambiguously match type representations and provide a more reliable test than comparing the string type names.

The `TypeCode` class, modeled after the corresponding CORBA API, provides access to type-code information. For details on the available operations for the `TypeCode` class, see the API Reference HTML documentation, which is available for all supported programming languages (select `Modules, RTI Connext`)
3.2 Built-in Data Types

**DDS API Reference, Topic Module, Type Code Support** or, for the Modern C++ API select **Modules, RTI Connext DDS API Reference, Infrastructure Module, DynamicType and DynamicData**.

**Note:** Type-code support must be enabled if you are going to use 5.4 ContentFilteredTopics on page 227 with the default SQL filter. You may disable type codes and use a custom filter, as described in 5.4.3 Creating ContentFilteredTopics on page 229.

### 3.1.3.1 Sending Type Information on the Network

In addition to being used locally, the type information of a *Topic* is published automatically during discovery as part of the built-in topics for publications and subscriptions. See 18.2 Built-in DataReaders on page 832. This allows applications to publish or subscribe to topics of arbitrary types. This functionality is useful for generic system monitoring tools like the *rtiddspy* debug tool (see the API Reference HTML documentation).

Earlier versions of *Connext DDS* (4.5f and lower) used serialized TypeCodes as the wire representation to communicate types over the network.

The OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3 uses TypeObjects as the wire representation. Types are propagated by serializing the associated TypeObject representation. *Connext DDS* 5.x and higher supports TypeObjects as the wire representation. To maintain backward compatibility with previous releases, *Connext DDS* still supports propagation of TypeCodes; however, support for this feature may be discontinued in future releases.

If your data type has an especially complex type code, you may need to increase the value of the `type_code_max_serialized_length`, `type_object_max_serialized_length`, and `type_object_max_deserialized_length` fields in the *DomainParticipant*'s 9.5.4 **DOMIAN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension)** on page 654. Or, to prevent the propagation of type information altogether, you can set these values to zero (0). Be aware that some features of monitoring tools, as well as some features of the middleware itself (such as ContentFilteredTopics) will not work correctly if you disable type information propagation.

For additional information on TypeCode versus TypeObject as wire representation, as well as resource limits to configure the propagation, see *Type Representation*, in the *RTI Connext DDS Core Libraries Extensible Types Guide*.

### 3.2 Built-in Data Types

*Connext DDS* provides a set of standard types that are built into the middleware. These types can be used immediately; they do not require you to write IDL, use *RTI Code Generator (rtiddsgen)* (see 3.6 Using RTI Code Generator (rtiddsgen) on page 152), or use the dynamic type API (see 3.2.7 Managing Memory for Built-in Types on page 69).

The supported built-in types are **String**, **KeyedString**, **Octets**, and **KeyedOctets**. (The latter two types are called **Bytes** and **KeyedBytes**, respectively, on Java and .NET platforms.)
3.2.1 Registering Built-in Types

The built-in type API is located under the DDS namespace in Traditional C++ and .NET. For Java, the API is contained inside the package `com.rti.dds.type.builtin`. In the Modern C++ API they are located in the dds::core namespace.

Built-in data types are discussed in the following sections.

### 3.2.1 Registering Built-in Types

By default, the built-in types are automatically registered when a `DomainParticipant` is created. You can change this behavior by setting the `DomainParticipant`'s `dds.builtin_type.auto_register` property to 0 (false) using the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.

### 3.2.2 Creating Topics for Built-in Types

To create a topic for a built-in type, just use the standard `DomainParticipant` operations, `create_topic()` or `create_topic_with_profile()` (see 5.1.1 Creating Topics on page 217); for the `type_name` parameter, use the value returned by the `get_type_name()` operation, listed below for each API.

**Note:** In the following examples, you will see the sentinel "<BuiltinType>.",

For C and Traditional C++: `<BuiltinType> = String, KeyedString, Octets or KeyedOctets

For Java and .NET¹: `<BuiltinType> = String, KeyedString, Bytes or KeyedBytes

**C API:**

```c
const char* DDS_<BuiltinType>Typesupport_get_type_name();
```

**Traditional C++ API with namespace:**

```c
const char* DDS::<BuiltinType>Typesupport::get_type_name();
```

**Traditional C++ API without namespace:**

```c
const char* DDS<BuiltinType>Typesupport::get_type_name();
```

**C++/CLI API:**

```c
System::String^ DDS:<BuiltinType>Typesupport::get_type_name();
```

**C# API:**

```c
System.String DDS.<BuiltinType>Typesupport.get_type_name();
```

**Java API:**

```c
String
com.rti.dds.type.builtin.<BuiltinType>Typesupport.get_type_name();
```

(This step is not required in the Modern C++ API)

---

¹RTI Connext DDS .NET language binding is currently supported for C# and C++/CLI.
3.2.2 Creating Topics for Built-in Types

3.2.2.1 Topic Creation Examples

For simplicity, error handling is not shown in the following examples.

C Example:

```c
DDS_Topics * topic = NULL;
/* Create a builtin type Topic */
topic = DDS_DomainParticipant_create_topic(
    participant, "StringTopic",
    DDS_StringTypeSupport_get_type_name(),
    DDS_TOPIC_QOS_DEFAULT, NULL,
    DDS_STATUS_MASK_NONE);
```

Traditional C++ Example with namespaces:¹

```cpp
using namespace DDS;
...
/* Create a String builtin type Topic */
Topic * topic = participant->create_topic(
    "StringTopic", StringTypeSupport::get_type_name(),
    DDS_TOPIC_QOS_DEFAULT, NULL, DDS_STATUS_MASK_NONE);
```

Modern C++ Example:

```cpp
dds::topic::Topic<dds::core::StringTopicType> topic(participant, "StringTopic");
```

C++/CLI Example:

```cpp
using namespace DDS;
...
/* Create a builtin type Topic */
Topic^ topic = participant->create_topic(
    "StringTopic", StringTypeSupport::get_type_name(),
    DomainParticipant::TOPIC_QOS_DEFAULT,
    nullptr, StatusMask::STATUS_MASK_NONE);
```

C# Example:

```csharp
using namespace DDS;
...
Create a builtin type Topic */
Topic topic = participant.create_topic(
    "StringTopic", StringTypeSupport.get_type_name(),
    DomainParticipant.TOPIC_QOS_DEFAULT,
    null, StatusMask.STATUS_MASK_NONE);
```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
Java Example:

```java
import com.rti.dds.type.builtin.*;
...
/* Create a builtin type Topic */
Topic topic = participant.create_topic(
    "StringTopic", StringTypeSupport.get_type_name(),
    DomainParticipant.TOPIC_QOS_DEFAULT,
    null, StatusKind.STATUS_MASK_NONE);
```

### 3.2.3 String Built-in Type

The String built-in type is represented by a NULL-terminated character array (char *) in C and C++ and an immutable String object in Java and .NET\(^1\). This type can be used to publish and subscribe to a single string.

#### 3.2.3.1 Creating and Deleting Strings

In C and C++, *Connext DDS* provides a set of operations to create (*DDS::String_alloc()*), destroy (*DDS::String_free()*), and clone strings (*DDS::String_dup()*). Select Modules, RTI Connext DDS API Reference, Infrastructure Module, String support in the API Reference HTML documentation, which is available for all supported programming languages.

**Memory Considerations in Copy Operations:**

When the read/take operations that take a sequence of strings as a parameter are used in copy mode, *Connext DDS* allocates the memory for the string elements in the sequence if they are initialized to NULL.

If the elements are not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory associated with the elements is reallocated with every DDS sample, because strings are immutable objects.
- In C and C++, the memory associated with the elements must be large enough to hold the received data. Insufficient memory may result in crashes.

When *take_next_sample()* and *read_next_sample()* are called in C and C++, you must make sure that the input string has enough memory to hold the received data. Insufficient memory may result in crashes.

#### 3.2.3.2 String DataWriter

The string *DataWriter* API matches the standard *DataWriter* API (see 7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308). There are no extensions.

---

\(^1\) *Connext DDS* .NET language binding is currently supported for C# and C++/CLI.
3.2.3 String Built-in Type

The following examples show how to write simple strings with a string built-in type *DataWriter*. For simplicity, error handling is not shown.

**C Example:**

```c
DDS_StringDataWriter * stringWriter = ...;
DDS_ReturnCode_t retCode; char * str = NULL;
/* Write some data */
retCode = DDS_StringDataWriter_write(
    stringWriter, "Hello World!", &DDS_HANDLE_NIL);
str = DDS_String_dup("Hello World!");
retCode = DDS.StringDataWriter_write(
    stringWriter, str, &DDS_HANDLE_NIL);
DDS.String_free(str);
```

**Traditional C++ Example with namespaces:**

```cpp
#include "ndds/ndds_namespace_cpp.h"

using namespace DDS;
...
StringDataWriter * stringWriter = ...;
/* Write some data */
ReturnCode_t retCode = stringWriter->write(
    "Hello World!", HANDLE_NIL);
char * str = DDS::String_dup("Hello World!");
retCode = stringWriter->write(str, HANDLE_NIL);
DDS::String_free(str);
```

**Modern C++ Example:**

```cpp
dds::pub::DataWriter<dds::core::StringTopicType> string_writer(
    participant, string_topic);
string_writer.write("Hello World!");
dds::core::string str = "Hello World!";
string_writer.write(str);
```

**C++/CLI Example:**

```cpp
using namespace System;
using namespace DDS;
...
StringDataWriter^ stringWriter = ...;
/* Write some data */
stringWriter->write(
    "Hello World!", InstanceHandle_t::HANDLE_NIL);
String^ str = "Hello World!";
stringWriter->write(
    str, InstanceHandle_t::HANDLE_NIL);
```

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
C# Example:

```csharp
using System;
using DDS;
...
StringDataWriter stringWriter = ...;
/* Write some data */
stringWriter.write(
    "Hello World!", InstanceHandle_t.HANDLENIL);
String str = "Hello World!";
stringWriter.write(
    str, InstanceHandle_t.HANDLENIL);
```

Java Example:

```java
import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
StringDataWriter stringWriter = ...;
/* Write some data */
stringWriter.write(
    "Hello World!", InstanceHandle_t.HANDLENIL);
String str = "Hello World!";
stringWriter.write(
    str, InstanceHandle_t.HANDLENIL);
```

### 3.2.3.3 String DataReader

The string `DataReader` API matches the standard `DataReader` API (see 8.4.1 Using a Type-Specific `DataReader` on page 553). There are no extensions.

The following examples show how to read simple strings with a string built-in type `DataReader`. For simplicity, error handling is not shown.
C Example:

```c
struct DDS_StringSeq dataSeq =
    DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq =
    DDS_SEQUENCE_INITIALIZER;
DDS_StringDataReader * stringReader = ... ;
DDS_ReturnCode_t retCode;
int i;
/* Take and print the data */
retCode = DDS_StringDataReader_take(
    stringReader, &dataSeq,
    &infoSeq, DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);
for (i = 0; i < DDS_StringSeq_get_length(&dataSeq);
    ++i) {
    if (DDS_SampleInfoSeq_get_reference(
        &info_seq, i)->valid_data) {
        DDS_StringTypeSupport_print_data(
            DDS_StringSeq_get(&data_seq, i));
    }
}
/* Return loan */
retCode = DDS_StringDataReader_return_loan(
    stringReader, &data_seq, &info_seq);
```

Traditional C++ Example with namespaces:¹

```cpp
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
StringSeq dataSeq;
SampleInfoSeq infoSeq;
StringDataReader * stringReader = ... ;
/* Take a print the data */
ReturnType retCode = stringReader->take(
    dataSeq, infoSeq,
    LENGTH_UNLIMITED,
    ANY_SAMPLE_STATE,
    ANY_VIEW_STATE,
    ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq[i].valid_data) {
        StringTypeSupport::print_data(dataSeq[i]);
    }
}
/* Return loan */
```

¹This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
3.2.3 String Built-in Type

```c++
retCode = stringReader->return_loan(
dataSeq, infoSeq);
```

**Modern C++ Example:**

```c++
using namespace dds::core;
using namespace dds::sub;
DataReader<StringTopicType> string_reader(
    participant, string_topic);
LoanedSamples<StringTopicType> samples =
    string_reader.take();
for (auto sample : samples) {
    if (sample.info().valid()) {
        std::cout << sample.data() << std::endl;
    }
}
```

**C++/CLI Example:**

```c++
using namespace System;
using namespace DDS;
...
StringSeq^ dataSeq = gcnew StringSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
StringDataReader^ stringReader = ... ;
/* Take and print the data */
stringReader->take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq->get_at(i)->valid_data) {
        StringTypeSupport::print_data(
            dataSeq->get_at(i));
    }
}/* Return loan */
stringReader->return_loan(dataSeq, infoSeq);
```
3.2.4 KeyedString Built-in Type

The Keyed String built-in type is represented by a (key, value) pair, where key and value are strings. This type can be used to publish and subscribe to keyed strings. The language specific representations of the type are as follows:

C# Example:

```csharp
using System;
using DDS;
...
StringSeq dataSeq = new StringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
StringDataReader stringReader = ...;
/* Take and print the data */
stringReader.take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < dataSeq.length(); ++i) {
    if (infoSeq.get_at(i).valid_data) {
        StringTypeSupport.print_data(
            dataSeq.get_at(i));
    }
}
```

Java Example:

```java
import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
StringSeq dataSeq = new StringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
StringDataReader stringReader = ...;
/* Take and print the data */
stringReader.take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < dataSeq.length(); ++i) {
    if (((SampleInfo)infoSeq.get(i)).valid_data) {
        System.out.println((String)dataSeq.get(i));
    }
}
/* Return loan */
stringReader.return_loan(dataSeq, infoSeq);
```
3.2.4 KeyedString Built-in Type

C/Traditional C++ Representation (without namespaces):

```c
struct DDS_KeyedString {
    char * key;
    char * value;
};
```

Modern C++ Representation:

```c++
class dds::core::KeyedStringTopicType {
public:
    dds::core::string& key();
    dds::core::string& value();
    // ... see API documentation for full definition
};
```

C++/CLI Representation:

```c
namespace DDS {
    public ref struct KeyedString: {
        public:
            System::String^ key;
            System::String^ value;
            ...
    };
}
```

C# Representation:

```c
namespace DDS {
    public class KeyedString {
        public System.String key;
        public System.String value;
    };
}
```

Java Representation:

```c
namespace DDS {
    public class KeyedString {
        public System.String key;
        public System.String value;
    };
}
```

3.2.4.1 Creating and Deleting Keyed Strings

Connext DDS provides a set of constructors/destructors to create/destroy Keyed Strings. For details, see the API Reference HTML documentation, which is available for all supported programming languages (select Modules, RTI Connext DDS API Reference, Topic Module, Built-in Types).

If you want to manipulate the memory of the fields 'value' and 'key' in the KeyedString struct in C/C++, use the operations DDS::String_alloc(), DDS::String_dup(), and DDS::String_free(), as described in
3.2.4.2 Keyed String DataWriter

The keyed string DataWriter API is extended with the following methods (in addition to the standard methods described in 7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308):

```cpp
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::dispose(
    const char* key,
    const DDS::InstanceHandle_t* instance_handle);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::dispose_w_timestamp(
    const char* key,
    const DDS::InstanceHandle_t* instance_handle,
    const struct DDS::Time_t* source_timestamp);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::get_key_value(
    char* key,
    const DDS::InstanceHandle_t* handle);
DDS::InstanceHandle_t
DDS::KeyedStringDataWriter::lookup_instance(
    const char* key);
DDS::InstanceHandle_t
DDS::KeyedStringDataWriter::register_instance(
    const char* key);
DDS::InstanceHandle_t
DDS::KeyedStringDataWriter::register_instance_w_timestamp(
    const char* key,
    const struct DDS::Time_t* source_timestamp);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::unregister_instance(
    const char* key,
    const DDS::InstanceHandle_t* handle);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::unregister_instance_w_timestamp(
    const char* key,
    const DDS::InstanceHandle_t* handle,
    const struct DDS::Time_t* source_timestamp);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::write(
    const char* key,
    const char* str,
    const DDS::InstanceHandle_t* handle);
DDS::ReturnCode_t
DDS::KeyedStringDataWriter::write_w_timestamp(
    const char* key,
    const char* str,
    const DDS::InstanceHandle_t* handle,
    const struct DDS::Time_t* source_timestamp);
```

These operations are introduced to provide maximum flexibility in the format of the input parameters for the write and instance management operations. For additional information and a complete description of the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Infrastructure Module, String Support).
the operations, see the API Reference HTML documentation, which is available for all supported programming languages.

The following examples show how to write keyed strings using a keyed string built-in type `DataWriter` and some of the extended APIs. For simplicity, error handling is not shown.

**C Example:**

```c
DDS_KeyedStringDataWriter * stringWriter = ...;
DDS_ReturnCode_t retCode;
struct DDS_KeyedString * keyedStr = NULL;
char * str = NULL;

/* Write some data using the KeyedString structure */
keyedStr = DDS_KeyedString_new(255, 255);
strcpy(keyedStr->key, "Key 1");
strcpy(keyedStr->value, "Value 1");
retCode = DDS_KeyedStringDataWriter_write(
    stringWriter,
    keyedStr,
    &DDS_HANDLE_NIL);
DDS_KeyedString_delete(keyedStr);

/* Write some data using individual strings */
retCode = DDS_KeyedStringDataWriter_write_string_w_key(
    stringWriter, "Key 1",
    "Value 1", &DDS_HANDLE_NIL);
str = DDS_String_dup("Value 2");
retCode = DDS_KeyedStringDataWriter_write_string_w_key(
    stringWriter, "Key 1",
    str, &DDS_HANDLE_NIL);
DDS_String_free(str);
```

**C++ Example with Namespaces:**

```c++
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;

... KeyedStringDataWriter * stringWriter = ...;

/* Write some data using the KeyedString */
KeyedString * keyedStr = new KeyedString(255, 255);
strcpy(keyedStr->key, "Key 1");
strcpy(keyedStr->value, "Value 1");
ReturnCode_t retCode = stringWriter->write(
    keyedStr, HANDLE_NIL);
delete keyedStr;
```

---

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
C++/CLI Example:

```cpp
using namespace System;
using namespace DDS;
...
KeyedStringDataWriter^ stringWriter = ... ;
/* Write some data using the KeyedString */
KeyedString^ keyedStr = gcnew KeyedString();
keyedStr->key = "Key 1";
keyedStr->value = "Value 1";
stringWriter->write(
    keyedStr, InstanceHandle_t::HANDLE_NIL);
/* Write some data using individual strings */
stringWriter->write(
    "Key 1","Value 1",
    InstanceHandle_t::HANDLE_NIL);
String^ str = "Value 2";
stringWriter->write(
    "Key 1", str,
    InstanceHandle_t::HANDLE_NIL);
```

C# Example:

```csharp
using System;
using DDS;
...
KeyedStringDataWriter stringWriter = ... ;
/* Write some data using the KeyedString */
KeyedString keyedStr = new KeyedString();
keyedStr.key = "Key 1";
keyedStr.value = "Value 1";
stringWriter.write(
    keyedStr, InstanceHandle_t.HANDLE_NIL);
/* Write some data using individual strings */
stringWriter.write(
    "Key 1", "Value 1",
    InstanceHandle_t.HANDLE_NIL);
String str = "Value 2";
stringWriter.write(
    "Key 1", str,
    InstanceHandle_t.HANDLE_NIL);
```

Java Example:

```java
import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
KeyedStringDataWriter stringWriter = ... ;
/* Write some data using the KeyedString */
KeyedString keyedStr = new KeyedString();
keyedStr.key = "Key 1";
keyedStr.value = "Value 1";
stringWriter.write(
    keyedStr, InstanceHandle_t.HANDLE_NIL);
/* Write some data using individual strings */
stringWriter.write(
    "Key 1", "Value 1",
    InstanceHandle_t.HANDLE_NIL);
```
3.2.4 KeyedString Built-in Type

```c
"Key 1", "Value 1",
InstanceHandle_t.HANDLE_NIL);
String str = "Value 2";
stringWriter.write(
"Key 1", str,
InstanceHandle_t.HANDLE_NIL);
```

3.2.4.3 Keyed String DataReader

The KeyedString DataReader API is extended with the following operations (in addition to the standard methods described in 8.4.1 Using a Type-Specific DataReader (FooDataReader) on page 553):

```c
DDS::ReturnCode_t
DDS::KeyedStringDataReader::get_key_value(
  char * key,
  const DDS::InstanceHandle_t* handle);
DDS::InstanceHandle_t
DDS::KeyedStringDataReader::lookup_instance(
  const char * key);
```

For additional information and a complete description of these operations in all supported languages, see the API Reference HTML documentation, which is available for all supported programming languages.

**Memory considerations in copy operations:**

For read/take operations with copy semantics, such as `read_next_sample()` and `take_next_sample()`, Connext DDS allocates memory for the fields 'value' and 'key' if they are initialized to NULL.

If the fields are not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory associated to the fields 'value' and 'key' will be reallocated with every DDS sample.
- In C and C++, the memory associated with the fields 'value' and 'key' must be large enough to hold the received data. Insufficient memory may result in crashes.

The following examples show how to read keyed strings with a keyed string built-in type DataReader. For simplicity, error handling is not shown.

**C Example:**

```c
struct DDS_KeyedStringSeq dataSeq =
  DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq =
  DDS_SEQUENCE_INITIALIZER;
DDS_KeyedKeyedStringDataReader * stringReader = ... ;
DDS_ReturnCode_t retCode;
int i;
/* Take and print the data */
retCode = DDS_KeyedStringDataReader_take(
  stringReader,
  &dataSeq,
  &infoSeq,
  DDS_LENGTH_UNLIMITED,
```
3.2.4 KeyedString Built-in Type

```c++
for (i = 0;
    i < DDS_KeyedStringSeq_get_length(&data_seq);
   ++i) {
     if (DDS_SampleInfoSeq_get_reference(
              &info_seq, i)->valid_data) {
       DDS_KeyedStringTypeSupport_print_data(
          DDS_KeyedStringSeq_get_reference(&data_seq, i));
     }
   }
*/
retCode = DDS_KeyedStringDataReader_return_loan(
    stringReader, &data_seq, &info_seq);
```

**C++ Example with Namespaces:**

```c++
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedStringSeq dataSeq;
SampleInfoSeq infoSeq;
KeyedStringDataReader * stringReader = ...;
/* Take a print the data */
ReturnCode_t retCode = stringReader->take(
    dataSeq, infoSeq,
    LENGTH_UNLIMITED,
    ANY_SAMPLE_STATE,
    ANY_VIEW_STATE,
    ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
  if (infoSeq[i].valid_data) {
    KeyedStringTypeSupport::print_data(&dataSeq[i]);
  }
}
/* Return loan */
retCode = stringReader->return_loan(dataSeq, infoSeq);
```

**C++/CLI Example:**

```c++
using namespace System;
using namespace DDS;
...
KeyedStringSeq^ dataSeq = gcnew KeyedStringSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
KeyedStringDataReader^ stringReader = ...;
/* Take and print the data */
stringReader->take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
```

---

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq->get_at(i)->valid_data) {
        KeyedStringTypeSupport::print_data(  
            dataSeq->get_at(i));  
    }
}
/* Return loan */
stringReader->return_loan(dataSeq, infoSeq);

C# Example:
using System;
using DDS;
...
KeyedStringSeq dataSeq = new KeyedStringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedStringDataReader stringReader = ... ;
/* Take and print the data */
stringReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq.get_at(i).valid_data) {
        KeyedStringTypeSupport.print_data(  
            dataSeq.get_at(i));  
    }
}
/* Return loan */
stringReader.return_loan(dataSeq, infoSeq);

Java Example:
import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
KeyedStringSeq dataSeq = new KeyedStringSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedStringDataReader stringReader = ... ;
/* Take and print the data */
stringReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (((SampleInfo)infoSeq.get(i)).valid_data) {
        System.out.println((  
            (KeyedString)dataSeq.get(i)).toString());  
    }
}
/* Return loan */
stringReader.return_loan(dataSeq, infoSeq);
3.2.5 Octets Built-in Type

The octets built-in type is used to send sequences of octets. The language-specific representations are as follows:

C/Traditional C++ Representation (without Namespaces):

```c
struct DDS_Octets {
    int length;
    unsigned char * value;
};
```

Modern C++ Representation:

```cpp
class dds::core::BytesTopicType {
public:
    uint8_t& operator [](uint32_t index);
    // ... see API documentation for full definition
};
```

C++/CLI Representation:

```csharp
namespace DDS {
    public ref struct Bytes { 
        public: 
            System::Int32 length; 
            System::Int32 offset; 
            array<System::Byte>^ value; 
            ... 
    }; 
};
```

C# Representation:

```csharp
namespace DDS {
    public class Bytes { 
        public System.Int32 length; 
        public System.Int32 offset; 
        public System.Byte[] value; 
        ... 
    }; 
};
```

Java Representation:

```java
package com.rti.dds.type.builtin;
public class Bytes implements Copyable {
    public int length;
    public int offset;
    public byte[] value;
    ... 
};
```
3.2.5 Octets Built-in Type

### 3.2.5.1 Creating and Deleting Octets

*Connext DDS* provides a set of constructors/destructors to create and destroy Octet objects. For details, see the API Reference HTML documentation, which is available for all supported programming languages (select Modules, RTI Connext DDS API Reference, Topic Module, Built-in Types).

If you want to manipulate the memory of the value field inside the Octets struct in C/Traditional C++, use the operations DDS::OctetBuffer_alloc(), DDS::OctetBuffer_dup(), and DDS::OctetBuffer_free(), described in the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Infrastructure Module, Octet Buffer Support).

### 3.2.5.2 Octets DataWriter

(Note: for Modern C++ API, refer to the API documentation)

In addition to the standard methods (see 7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308), the octets *DataWriter* API is extended with the following methods:

```c
DDS::ReturnCode_t DDS::OctetsDataWriter::write(
    const DDS::OctetSeq & octets,
    const DDS::InstanceHandle_t & handle);
DDS::ReturnCode_t DDS::OctetsDataWriter::write(
    const unsigned char * octets,
    int length,
    const DDS::InstanceHandle_t & handle);
DDS::ReturnCode_t DDS::OctetsDataWriter::write_w_timestamp(
    const DDS::OctetSeq & octets,
    const DDS::InstanceHandle_t & handle,
    const DDS::Time_t & source_timestamp);
DDS::ReturnCode_t DDS::OctetsDataWriter::write_w_timestamp(
    const unsigned char * octets,
    int length,
    const DDS::InstanceHandle_t & handle,
    const DDS::Time_t & source_timestamp);
```

These methods are introduced to provide maximum flexibility in the format of the input parameters for the write operations. For additional information and a complete description of these operations in all supported languages, see the API Reference HTML documentation.

The following examples show how to write an array of octets using an octets built-in type *DataWriter* and some of the extended APIs. For simplicity, error handling is not shown.
C Example:

```c
DDS_OctetsDataWriter * octetsWriter = ...;
DDS_ReturnCode_t retCode;
struct DDS_Octets * octets = NULL;
char * octetArray = NULL;
/* Write some data using the Octets structure */
octets = DDS_Octets_new_w_size(1024);
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
retCode = DDS_OctetsDataWriter_write(
    octetsWriter, octets, &DDS_HANDLE_NIL);
DDS_Octets_delete(octets);
/* Write some data using an octets array */
octetArray = (unsigned char *)malloc(1024);
octetArray[0] = 46;
octetArray[1] = 47;
retCode = DDS_OctetsDataWriter_write_octets(
    octetsWriter, octetArray, 2,
    &DDS_HANDLE_NIL);
free(octetArray);
```

C++ Example with Namespaces:

```cpp
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
OctetsDataWriter * octetsWriter = ...;
/* Write some data using the Octets structure */
Octets * octets = new Octets(1024);
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
ReturnCode_t retCode = octetsWriter->write(octets, HANDLE_NIL);
delete octets;
/* Write some data using an octet array */
unsigned char * octetArray = new unsigned char[1024];
octetArray[0] = 46;
octetArray[1] = 47;
retCode = octetsWriter->write(octetArray, 2, HANDLE_NIL);
delete []octetArray;
```

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
C++/CLI Example:

```c++
using namespace System;
using namespace DDS;
...
BytesDataWriter^ octetsWriter = ...;
//* Write some data using Bytes */
Bytes^ octets = gcnew Bytes(1024);
octets->value[0] = 46;
octets->value[1] = 47;
octets.length = 2;
octets.offset = 0;
octetsWriter->write(octets, InstanceHandle_t::HANDLE_NIL);
//* Write some data using individual strings */
array<Byte>^ octetArray = gcnew array<Byte>(1024);
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter->write(octetArray, 0, 2, InstanceHandle_t::HANDLE_NIL);
```

C# Example:

```csharp
using System;
using DDS;
...
BytesDataWriter stringWriter = ...;
//* Write some data using the Bytes */
Bytes octets = new Bytes(1024);
octets.value[0] = 46;
octets.value[1] = 47;
octets.length = 2;
octets.offset = 0;
octetsWriter.write(octets, InstanceHandle_t.HANDLE_NIL);
//* Write some data using individual strings */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter.write(octetArray, 0, 2, InstanceHandle_t.HANDLE_NIL);
```

Java Example:

```java
import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
BytesDataWriter octetsWriter = ...;
/* Write some data using the Bytes class*/
Bytes octets = new Bytes(1024);
octets.length = 2;
octets.offset = 0;
octets.value[0] = 46;
octets.value[1] = 47;
ocetWriter.write(octets, InstanceHandle_t.HANDLE_NIL);
/* Write some data using a byte array */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;
ocetWriter.write(octetArray, 0, 2, InstanceHandle_t.HANDLE_NIL);
```
3.2.5.3 Octets DataReader

(Note: for the Modern C++ API, refer to the API Reference HTML documentation)

The octets DataReader API matches the standard DataReader API (see 8.4.1 Using a Type-Specific DataReader (FooDataReader) on page 553). There are no extensions.

Memory considerations in copy operations:

For read/take operations with copy semantics, such as read_next_sample() and take_next_sample(), Connext DDS allocates memory for the field 'value' if it is initialized to NULL.

If the field 'value' is not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory for the field 'value' will be reallocated if the current size is not large enough to hold the received data.
- In C and C++, the memory associated with the field 'value' must be big enough to hold the received data. Insufficient memory may result in crashes.

The following examples show how to read octets with an octets built-in type DataReader. For simplicity, error handling is not shown.

C Example:

```c
struct DDS_OctetsSeq dataSeq = DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq = DDS_SEQUENCE_INITIALIZER;
DDS_OctetsDataReader * octetsReader = ...;
DDS_ReturnCode_t retCode;
int i;
/* Take and print the data */
retCode = DDS_OctetsDataReader_take(
    octetsReader, &dataSeq,
    &infoSeq, DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);
for (i = 0; i < DDS_OctetsSeq_get_length(&dataSeq); ++i) {
    if (DDS_SampleInfoSeq_get_reference(
        &infoSeq, i)->valid_data) {
        DDS_OctetsTypeSupport_print_data(
            DDS_OctetsSeq_get_reference(&dataSeq, i));
    }
}
/* Return loan */
retCode = DDS_OctetsDataReader_return_loan(
    octetsReader, &dataSeq, &infoSeq);
```
3.2.5 Octets Built-in Type

C++ Example with Namespaces:

```
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
OctetsSeq dataSeq;
SampleInfoSeq infoSeq;
OctetsDataReader * octetsReader = ...;
/* Take a print the data */
ReturnCode_t retCode = octetsReader->take(
    dataSeq, infoSeq,
    LENGTH_UNLIMITED, ANY_SAMPLE_STATE,
    ANY_VIEW_STATE, ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq[i].valid_data) {
        OctetsTypeSupport::print_data(&dataSeq[i]);
    }
}
/* Return loan */
retCode = octetsReader->return_loan(dataSeq, infoSeq);
```

C++/CLI Example:

```
using namespace System;
using namespace DDS;
...
BytesSeq^ dataSeq = gcnew BytesSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
BytesDataReader^ octetsReader = ...;
/* Take and print the data */
octetsReader->take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq->get_at(i)->valid_data) {
        BytesTypeSupport::print_data(dataSeq->get_at(i));
    }
}
/* Return loan */
octetsReader->return_loan(dataSeq, infoSeq);
```

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
C# Example:

```csharp
using System;
using DDS;
...
BytesSeq dataSeq = new BytesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
BytesDataReader octetsReader = ...
/* Take and print the data */
octetsReader.take(
    dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq.get_at(i)).valid_data) {
        BytesTypeSupport.print_data(dataSeq.get_at(i));
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);
```

Java Example:

```java
import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
BytesSeq dataSeq = new BytesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
BytesDataReader octetsReader = ...
/* Take and print the data */
octetsReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind.ANY_SAMPLE_STATE,
    ViewStateKind.ANY_VIEW_STATE,
    InstanceStateKind.ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (((SampleInfo)infoSeq.get(i)).valid_data) {
        System.out.println(((Bytes)dataSeq.get(i)).toString());
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);
```

3.2.6 KeyedOctets Built-in Type

The keyed octets built-in type is used to send sequences of octets with a key. The language-specific representations of the type are as follows:
C/Traditional C++ Representation (without Namespaces):

```cpp
struct DDS_KeyedOctets {
    char * key;
    int length;
    unsigned char * value;
};
```

Modern C++ Representation:

```cpp
class dds::core::KeyedStringTopicType {
    public:
        dds::core::string& key();
        uint8_t& operator [](uint32_t index);
    // ... see API documentation for full definition
};
```

C++/CLI Representation:

```cpp
namespace DDS {
    public ref struct KeyedBytes {
        public:
            System::String^ key;
            System::Int32 length;
            System::Int32 offset;
            System::Byte[] value;
            ...
    };
};
```

C# Representation:

```cs
namespace DDS {
    public class KeyedBytes {
        public System.String key;
        public System.Int32 length;
        public System.Int32 offset;
        public System.Byte[] value;
        ...
    };
};
```

Java Representation:

```java
package com.rti.dds.type.builtin;
public class KeyedBytes {
    public String key;
    public int length;
    public int offset;
    public byte[] value;
    ...
};
```
3.2.6 KeyedOctets Built-in Type

3.2.6.1 Creating and Deleting KeyedOctets

Connext DDS provides a set of constructors/destructors to create/destroy KeyedOctets objects. For details, see the API Reference HTML documentation, which is available for all supported programming languages (select Modules, RTI Connext DDS API Reference, Topic Module, Built-in Types).

To manipulate the memory of the value field in the KeyedOctets struct in C/C++: use DDS::OctetBuffer_alloc(), DDS::OctetBuffer_dup(), and DDS::OctetBuffer_free(). See the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Infrastructure Module, Octet Buffer Support).

To manipulate the memory of the key field in the KeyedOctets struct in C/C++: use DDS::String_alloc(), DDS::String_dup(), and DDS::String_free(). See the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Infrastructure Module, String Support).

3.2.6.2 Keyed Octets DataWriter

In addition to the standard methods (see 7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308), the keyed octets DataWriter API is extended with the following methods:

```c
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::dispose(
    const char* key,
    const DDS::InstanceHandle_t & instance_handle);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::dispose_w_timestamp(
    const char* key,
    const DDS::InstanceHandle_t & instance_handle,
    const DDS::Time_t & source_timestamp);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::get_key_value(
    char * key,
    const DDS::InstanceHandle_t& handle);
DDS::InstanceHandle_t
DDS::KeyedOctetsDataWriter::lookup_instance(
    const char * key);
DDS::InstanceHandle_t
DDS::KeyedOctetsDataWriter::register_instance(
    const char* key);
DDS::InstanceHandle_t
DDS::KeyedOctetsDataWriter::register_instance_w_timestamp(
    const char * key,
    const DDS::Time_t & source_timestamp);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::unregister_instance(
    const char * key,
    const DDS::InstanceHandle_t & handle);
DDS::ReturnCode_t
DDS::KeyedOctetsDataWriter::unregister_instance_w_timestamp(
    const char* key,
    const DDS::InstanceHandle_t & handle,
```
These methods are introduced to provide maximum flexibility in the format of the input parameters for the write and instance management operations. For more information and a complete description of these operations in all supported languages, see the API Reference HTML documentation.

The following examples show how to write keyed octets using a keyed octets built-in type DataWriter and some of the extended APIs. For simplicity, error handling is not shown.

**C Example:**

```c
DDS_KeyedOctetsDataWriter * octetsWriter = ...;
DDS_ReturnCode_t retCode;
struct DDS_KeyedOctets * octets = NULL;
char * octetArray = NULL;
/* Write some data using KeyedOctets structure */
octets = DDS_KeyedOctets_new_w_size(128,1024);
strcpy(octets->key, "Key 1");
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
retCode = DDS_KeyedOctetsDataWriter_write( 
octetsWriter, octets, &DDS_HANDLE_NIL);
DDS_KeyedOctets_delete(octets);
/* Write some data using an octets array */
octetArray = (unsigned char *)malloc(1024);
octetArray[0] = 46;
octetArray[1] = 47;
retCode = 
```
3.2.6 KeyedOctets Built-in Type

```c
DDS_KeyedOctetsDataWriter_write_octets_w_key (  
octetsWriter, "Key 1",  
octetArray, 2, &DDS_HANDLE_NIL);  
free(octetArray);
```

**C++ Example with Namespaces:**

```c
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedOctetsDataWriter * octetsWriter = ...;
/* Write some data using KeyedOctets */
KeyedOctets * octets = new KeyedOctets(128,1024);
strcpy(octets->key, "Key 1");
octets->length = 2;
octets->value[0] = 46;
octets->value[1] = 47;
ReturnCode_t retCode =  
    octetsWriter->write(octets, HANDLE_NIL);
delete octets;
/* Write some data using an octet array */
unsigned char * octetArray = new unsigned char[1024];
octetArray[0] = 46;
octetArray[1] = 47;
retCode = octetsWriter->write(  
    "Key 1", octetArray, 2, HANDLE_NIL);
delete []octetArray;
```

**C++/CLI Example:**

```c
using namespace System;
using namespace DDS;
...
KeyedOctetsDataWriter^ octetsWriter = ... ;
/* Write some data using KeyedBytes */
KeyedBytes^ octets = gcnew KeyedBytes(1024);
octets->key = "Key 1";
octets->value[0] =46;
octets->value[1] =47;
octets.length = 2;
octets.offset = 0;
octetWriter->write(  
    octets, InstanceHandle_t::HANDLE_NIL);
/* Write some data using individual strings */
```

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
3.2.6 KeyedOctets Built-in Type

```csharp
array<Byte>^ octetArray = gcnew array<Byte>(1024);
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter->write(
    "Key 1", octetArray,
    0, 2, InstanceHandle_t::HANDLE NIL);
```

**C# Example:**

```csharp
using System;
using DDS;
...
KeyedBytesDataWriter stringWriter = ...;
/* Write some data using the KeyedBytes */
KeyedBytes octets = new KeyedBytes(1024);
    octets.key = "Key 1";
    octets.value[0] = 46;
    octets.value[1] = 47;
    octets.length = 2;
    octets.offset = 0;
    octetWriter.write(octets,
        InstanceHandle_t::HANDLE NIL);
/* Write some data using individual strings */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter.write(
    "Key 1", octetArray,
    0, 2, InstanceHandle_t::HANDLE NIL);
```

**Java Example:**

```java
import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
KeyedBytesDataWriter octetsWriter = ...;
/* Write some data using KeyedBytes class */
KeyedBytes octets = new KeyedBytes(1024);
octets.key = "Key 1";
octets.length = 2;
octets.offset = 0;
octets.value[0] = 46;
octets.value[1] = 47;
octetsWriter.write(octets,
        InstanceHandle_t::HANDLE NIL);
/* Write some data using a byte array */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
```
3.2.6 KeyedOctets Built-in Type

octetArray[1] = 47;
octetsWriter.write(
    "Key 1", octetArray,
0, 2, InstanceHandle_t.HANDLE_NIL);

3.2.6.3 Keyed Octets DataReader

The KeyedOctets DataReader API is extended with the following methods (in addition to the standard methods described in 8.4.1 Using a Type-Specific DataReader (FooDataReader) on page 553):

```cpp
DDS::ReturnCode_t
DDS::KeyedOctetsDataReader::get_key_value(
    char * key,
    const DDS::InstanceHandle_t* handle);

DDS::InstanceHandle_t
DDS::KeyedOctetsDataReader::lookup_instance(
    char * key);
```

For more information and a complete description of these operations in all supported languages, see the API Reference HTML documentation.

**Memory considerations in copy operations:**

For read/take operations with copy semantics, such as `read_next_sample()` and `take_next_sample()`, Connext DDS allocates memory for the fields 'value' and 'key' if they are initialized to NULL.

If the fields are not initialized to NULL, the behavior depends on the language:

- In Java and .NET, the memory of the field 'value' will be reallocated if the current size is not large enough to hold the received data. The memory associated with the field 'key' will be reallocated with every DDS sample (the key is an immutable object).
- In C and C++, the memory associated with the fields 'value' and 'key' must be large enough to hold the received data. Insufficient memory may result in crashes.

The following examples show how to read keyed octets with a keyed octets built-in type DataReader. For simplicity, error handling is not shown.

**C Example:**

```c
struct DDS_KeyedOctetsSeq dataSeq =
    DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq =
    DDS_SEQUENCE_INITIALIZER;
DDS_KeyedOctetsDataReader * octetsReader = ... ;
DDS_ReturnCode_t retCode;
```
3.2.6 KeyedOctets Built-in Type

```cpp
int i;
/* Take and print the data */
retCode = DDS_KeyedOctetsDataReader_take(
    octetsReader,
    &dataSeq, &infoSeq, DDS_LENGTH_UNLIMITED,
    DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);
for (i = 0; i < DDS_KeyedOctetsSeq_get_length(&data_seq); ++i) {
    if (DDS_SampleInfoSeq_get_reference(
            &info_seq, i)->valid_data) {
        DDS_KeyedOctetsTypeSupport_print_data(
            DDS_KeyedOctetsSeq_get_reference(
                &data_seq, i));
    }
}
/* Return loan */
retCode = DDS_KeyedOctetsDataReader_return_loan(
    octetsReader, &data_seq, &info_seq);
```

C++ Example with Namespaces:

```cpp
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedOctetsSeq dataSeq;
SampleInfoSeq infoSeq;
KeyedOctetsDataReader * octetsReader = ...;
/* Take and print the data */
ReturnCode_t retCode = octetsReader->take(
    dataSeq, infoSeq, LENGTH_UNLIMITED,
    ANY_SAMPLE_STATE, ANY_VIEW_STATE,
    ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i) {
    if (infoSeq[i].valid_data) {
        KeyedOctetsTypeSupport::print_data(
            &dataSeq[i]);
    }
}
/* Return loan */
retCode = octetsReader->return_loan(
    dataSeq, infoSeq);
```

C++/CLI Example:

```cpp
using namespace System;
using namespace DDS;
...
KeyedBytesSeq^ dataSeq = gcnew KeyedBytesSeq();
SampleInfoSeq^ infoSeq = gcnew SampleInfoSeq();
KeyedBytesDataReader^ octetsReader = ...;
```

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
3.2.6 KeyedOctets Built-in Type

C# Example:

```csharp
using System;
using DDS;
...
KeyedBytesSeq dataSeq = new KeyedBytesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedBytesDataReader octetsReader = ...

/* Take and print the data */
octetsReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy::LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i){
    if (infoSeq->get_at(i)->valid_data){
        KeyedBytesTypeSupport::print_data(
            dataSeq->get_at(i));
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);
```

Java Example:

```java
import com.rti.dds.infrastructure.*;
import com.rti.dds.subscription.*;
import com.rti.dds.type.builtin.*;
...
KeyedBytesSeq dataSeq = new KeyedBytesSeq();
SampleInfoSeq infoSeq = new SampleInfoSeq();
KeyedBytesDataReader octetsReader = ...

/* Take and print the data */
octetsReader.take(dataSeq, infoSeq,
    ResourceLimitsQosPolicy.LENGTH_UNLIMITED,
    SampleStateKind::ANY_SAMPLE_STATE,
    ViewStateKind::ANY_VIEW_STATE,
    InstanceStateKind::ANY_INSTANCE_STATE);
for (int i = 0; i < data_seq.length(); ++i){
    if (((SampleInfo)infoSeq.get(i)).valid_data){
        System.out.println(
            dataSeq.get_at(i));
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);
```
3.2.7 Managing Memory for Built-in Types

When a DDS sample is written, the DataWriter serializes it and stores the result in a buffer obtained from a pool of preallocated buffers. In the same way, when a DDS sample is received, the DataReader deserializes it and stores the result in a DDS sample coming from a pool of preallocated DDS samples.

By default, the buffers on the DataWriter and the samples on the DataReader are preallocated with their maximum size. For example:

```java
struct MyString {
    string<128> value;
};
```

This IDL-defined type has a maximum serialized size of 133 bytes (4 bytes for length + 128 characters + 1 NULL terminating character). So the serialization buffers will have a size of 133 bytes. The buffer can hold samples with 128 characters strings. Consequently, the preallocated samples will be sized to keep this length.

However, for built-in types, the maximum size of the buffers/DDS samples is unknown and depends on the nature of the application using the built-in type.

For example, a video surveillance application that is using the keyed octets built-in type to publish a stream of images will require bigger buffers than a market-data application that uses the same built-in type to publish market-data values.

To accommodate both kinds of applications and optimize memory usage, you can configure the maximum size of the built-in types on a per-DataWriter or per-DataReader basis using the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439. Table 3.1 Properties for Allocating Size of Built-in Types, per DataWriter and DataReader lists the supported built-in type properties. When the properties are defined in the DomainParticipant, they are applicable to all DataWriters and DataReaders belonging to the DomainParticipant, unless they are overwritten in the DataWriters and DataReaders.

These properties must be set consistently with respect to the corresponding *max_size* properties in the DomainParticipant (see Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant). The value of the alloc_size property must be less than or equal to the max_size property with the same name prefix in the DomainParticipant.

Unbounded built-in types are only supported in the C, C++, Java, and .NET APIs.

3.2.7.1 Examples—Setting the Maximum Size for a String Programmatically on page 71 includes examples of how to set the maximum size of a string built-in type for a DataWriter programmatically, for
each API. You can also set the maximum size of the built-in types using XML QoS Profiles. For example, the following XML shows how to set the maximum size of a string built-in type for a DataWriter.

```xml
<dds>
<qos_library name="BuiltinExampleLibrary">
  <qos_profile name="BuiltinExampleProfile">
    <datawriter_qos>
      <property>
        <value>
          <element>
            <name>dds.builtin_type.string.alloc_size</name>
            <value>2048</value>
          </element>
        </value>
      </property>
    </datawriter_qos>
    <datareader_qos>
      <property>
        <value>
          <element>
            <name>dds.builtin_type.string.alloc_size</name>
            <value>2048</value>
          </element>
        </value>
      </property>
    </datareader_qos>
  </qos_profile>
</qos_library>
</dds>
```

**Table 3.1 Properties for Allocating Size of Built-in Types, per DataWriter and DataReader**

<table>
<thead>
<tr>
<th>Built-in Type</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>dds.builtin_type.string.alloc_size</td>
<td>Maximum size of the strings published by the DataWriter or received by the DataReader (includes the NULL-terminated character). Default: dds.builtin_type.string.max_size if defined (see Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.</td>
</tr>
<tr>
<td>keyed_string</td>
<td>dds.builtin_type.keyed_string.alloc_key_size</td>
<td>Maximum size of the keys used by the DataWriter or DataReader (includes the NULL-terminated character). Default: dds.builtin_type.keyed_string.max_key_size if defined (see Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.</td>
</tr>
<tr>
<td></td>
<td>dds.builtin_type.keyed_string.alloc_size</td>
<td>Maximum size of the strings published by the DataWriter or received by the DataReader (includes the NULL-terminated character). Default: dds.builtin_type.keyed_string.max_size if defined (see Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.</td>
</tr>
</tbody>
</table>
### Table 3.1 Properties for Allocating Size of Built-in Types, per DataWriter and DataReader

<table>
<thead>
<tr>
<th>Built-in Type</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>octets</td>
<td>dds.builtin_type.octets.alloc_size</td>
<td>Maximum size of the octet sequences published by the DataWriter or DataReader. Default: dds.builtin_type.octets.max_size if defined (see Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 2048.</td>
</tr>
<tr>
<td>keyed-octets</td>
<td>dds.builtin_type.keyed_octets.alloc_key_size</td>
<td>Maximum size of the key published by the DataWriter or received by the DataWriter (includes the NULL-terminated character). Default: dds.builtin_type.keyed_octets.max_key_size if defined (see Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 1024.</td>
</tr>
<tr>
<td></td>
<td>dds.builtin_type.keyed_octets.alloc_size</td>
<td>Maximum size of the octet sequences published by the DataWriter or DataReader. Default: dds.builtin_type.keyed_octets.max_size if defined (see Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant). Otherwise, 2048.</td>
</tr>
</tbody>
</table>

### 3.2.7.1 Examples—Setting the Maximum Size for a String Programmatically

For simplicity, error handling is not shown in the following examples.

**C Example:**

```c
DDS_DataWriter * writer = NULL;
DDS_StringDataWriter * stringWriter = NULL;
DDS_Publisher * publisher = ...
DDS_Topic * stringTopic = ...
struct DDS_DataWriterQos writerQos =
    DDS_DataWriterQos_INITIALIZER;
DDS_ReturnCode_t retCode;
retCode = DDS_DomainParticipant_get_default_datawriter_qos (participant, &writerQos);
retCode = DDS_PropertyQosPolicyHelper_add_property (writerQos.property, "dds.builtin_type.string.alloc_size", "1000", DDS_BOOLEAN_FALSE);
writer = DDS_Publisher_create_datawriter(publisher, stringTopic, &writerQos, NULL, DDS_STATUS_MASK_NONE);
stringWriter = DDS_StringDataWriter_narrow(writer);
DDS_DataWriterQos_finalize(&writerQos);
```

**Traditional C++ Example with Namespaces:**

```cpp
#include "ndds/ndds_namespace_cpp.h"
```

---

1This example uses C++ namespaces. If you're not using namespaces in your own code, prefix the name of each DDS class with 'DDS.' For example, DDS::StringDataWriter becomes DDSStringDataWriter.
using namespace DDS;
...
Publisher * publisher = ... ;
Topic * stringTopic = ... ;
DataWriterQos writerQos;
ReturnCode_t retCode =
    participant->get_default_datawriter_qos(writerQos);
retCode = PropertyQosPolicyHelper::add_property ( 
    &writerQos.property,
    "dds.builtin_type.string.alloc_size",
    "1000", BOOLEAN_FALSE);
DataWriter * writer = publisher->create_datawriter(
    stringTopic, writerQos,
    NULL, STATUS_MASK_NONE);
StringDataWriter * stringWriter =
    StringDataWriter::narrow(writer);

Modern C++ Example:

dds::pub::qos::DataWriterQos writer_qos =
    participant.default_datawriter_qos();
writer_qos.policy<rti::core::policy::Property>() .set( { 
    "dds.builtin_type.string.alloc_size", "1000" });
dds::pub::DataWriter<dds::core::StringTopicType> writer( 
    publisher, string_topic, writer_qos);

C++/CLI Example:

using namespace DDS;
...
Topic^ stringTopic = ... ;
Publisher^ publisher = ... ;
DataWriterQos^ writerQos = gcnew DataWriterQos();
participant->get_default_datawriter_qos(writerQos);
PropertyQosPolicyHelper::add_property( 
    writerQos->property_qos,
    "dds.builtin_type.string.alloc_size",
    "1000", false);
DataWriter^ writer = publisher->create_datawriter(
    stringTopic, writerQos,
    nullptr, StatusMask::STATUS_MASK_NONE);
StringDataWriter^ stringWriter =
    safe_cast<StringDataWriter^>(writer);

C# Example:

using DDS;
...

3.2.7 Managing Memory for Built-in Types

Java Example:

```java
import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...
Topic stringTopic = ... ;
Publisher publisher = ... ;
DataWriterQos writerQos = new DataWriterQos();
participant.get_default_datawriter_qos(writerQos);
PropertyQosPolicyHelper.add_property (writerQos.property,
   "dds.builtin_type.string.alloc_size", 
   "1000", false);
StringDataWriter stringWriter = (StringDataWriter) publisher.create_datawriter( 
   stringTopic, writerQos, null, StatusMask.STATUS_MASK_NONE);
```

3.2.7.2 Unbounded Built-in Types

In some scenarios, the maximum size of a built-in type is not known in advance and there is no reasonable maximum size. For example, this could occur in a file transfer application using the built-in type Octets. Setting a large value for the `dds.builtin_type.*.alloc_size` property would involve high memory usage.

**Note:** Replace `*` with one of the built-in type names. See Table 3.1 Properties for Allocating Size of Built-in Types, per DataWriter and DataReader for the full property names.

For the above use case, you can configure the built-in type to be unbounded by setting the property `dds.builtin_type.*.alloc_size` to the maximum value of a 32-bit signed integer: 2,147,483,647. Then the middleware will not preallocate the `DataReader` queue's samples to their maximum size. Instead, it will deserialize incoming samples by dynamically allocating and deallocating memory to accommodate the actual size of the sample value.
To configure unbounded support for built-in types:

1. Use these threshold QoS properties:
   - `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size` on the `DataWriter`
   - `dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size` on the `DataReader`

2. Set the QoS value `reader_resource_limits.dynamically_allocate_fragmented_samples` on the `DataReader` to true.

3. For the Java API, also set these properties accordingly for the Java serialization buffer:
   - `dds.data_writer.history.memory_manager.java_stream.min_size`
   - `dds.data_writer.history.memory_manager.java_stream.trim_to_size`
   - `dds.data_reader.history.memory_manager.java_stream.min_size`
   - `dds.data_reader.history.memory_manager.java_stream.trim_to_size`

See these sections:

- 22.1.3 Writer-Side Memory Management when Using Java on page 931
- 22.2.2 Reader-Side Memory Management when Using Java on page 935

Unbounded built-in types are only supported in the C, C++, .NET, and Java APIs.

### 3.2.8 Type Codes for Built-in Types

The type codes associated with the built-in types are generated from the following IDL type definitions:

```idl
module DDS {
    /* String */
    struct String {
        string<max_size> value;
    };
    /* KeyedString */
    struct KeyedString {
        @key string<max_size> key;
        string<max_size> value;
    };
    /* Octets */
    struct Octets {
        sequence<octet, max_size> value;
    };
    /* KeyedOctets */
    struct KeyedOctets {
        @key string<max_size> key;
        sequence<octet, max_size> value;
    }
}
```
The maximum size (**max_size**) of the strings and sequences that will be included in the type code definitions can be configured on a per-**DomainParticipant**-basis by using the properties in **Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant**.

**Table 3.2 Properties for Allocating Size of Built-in Types, per DomainParticipant**

<table>
<thead>
<tr>
<th>Built-in Type</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>dds.builtin_type.string.max_size</td>
<td>Maximum size of the strings published by the <strong>DataWriters</strong> and received by the <strong>DataReaders</strong> belonging to a <strong>DomainParticipant</strong> (includes the NULL-terminated character). Default: 1024</td>
</tr>
<tr>
<td>KeyedString</td>
<td>dds.builtin_type.keyed_string_max_key_size</td>
<td>Maximum size of the keys used by the <strong>DataWriters</strong> and <strong>DataReaders</strong> belonging to a <strong>DomainParticipant</strong> (includes the NULL-terminated character). Default: 1024</td>
</tr>
<tr>
<td>Octets</td>
<td>dds.builtin_type.octets.max_size</td>
<td>Maximum size of the octet sequences published by the <strong>DataWriters</strong> and <strong>DataReaders</strong> belonging to a <strong>DomainParticipant</strong>. Default: 2048</td>
</tr>
<tr>
<td>Keyed-Octets</td>
<td>dds.builtin_type.keyed_octets_max_key_size</td>
<td>Maximum size of the key published by the <strong>DataWriter</strong> and received by the <strong>DataReaders</strong> belonging to the <strong>DomainParticipant</strong> (includes the NULL-terminated character). Default: 1024.</td>
</tr>
<tr>
<td></td>
<td>dds.builtin_type.keyed_octets_max_size</td>
<td>Maximum size of the octet sequences published by the <strong>DataWriters</strong> and <strong>DataReaders</strong> belonging to a <strong>DomainParticipant</strong>. Default: 2048</td>
</tr>
</tbody>
</table>

### 3.3 Creating User Data Types with IDL

You can create user data types in a text file using **IDL** (Interface Description Language). **IDL** is programming-language independent, so the same file can be used to generate code in C, Traditional C++, Modern C++, C++/CLI, Ada, and Java (the languages supported by **RTI Code Generator** (rtiddsgen)). **RTI Code Generator** parses the **IDL** file and automatically generates all the necessary routines and wrapper functions to bind the types for use by **Connext DDS** at run time. You will end up with a set of required routines and structures that your application and **Connext DDS** will use to manipulate the data.

**Connext DDS** only uses a subset of the **IDL** 4.2 ([https://www.omg.org/spec/IDL](https://www.omg.org/spec/IDL)) syntax. **IDL** was originally defined by the OMG for the use of CORBA client/server applications in an enterprise setting. Not all of the constructs that can be described by the language are as useful in the context of high-performance...
3.3 Creating User Data Types with IDL

data-centric embedded applications. These include the constructs that define method and function prototypes like “interface.”

*RTI Code Generator* will parse any file that follows version 4.2 of the IDL specification. It will ignore and show a warning for all syntax that is not recognized by *Connext DDS*. There is a limit of 256 characters for the length of a variable name in an IDL file.

Certain keywords are considered reserved by the IDL specification; see Table 3.3 Reserved IDL Keywords.

**Note:** Table 3.3 Reserved IDL Keywords does not include other words that may be used by macros for different compilers and operating systems. For example, *min* and *max* are reserved key words for Microsoft Visual Studio 2015.

<table>
<thead>
<tr>
<th>Table 3.3 Reserved IDL Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
</tr>
<tr>
<td>bitmask</td>
</tr>
<tr>
<td>component</td>
</tr>
<tr>
<td>custom</td>
</tr>
<tr>
<td>enum</td>
</tr>
<tr>
<td>fixed</td>
</tr>
<tr>
<td>in</td>
</tr>
<tr>
<td>manages</td>
</tr>
<tr>
<td>native</td>
</tr>
<tr>
<td>primarykey</td>
</tr>
<tr>
<td>public</td>
</tr>
<tr>
<td>sequence</td>
</tr>
<tr>
<td>switch</td>
</tr>
<tr>
<td>typename</td>
</tr>
<tr>
<td>ValueBase</td>
</tr>
<tr>
<td>int8</td>
</tr>
<tr>
<td>uint16</td>
</tr>
</tbody>
</table>

The IDL constructs supported by *RTI Code Generator* are described in Table 3.5 Specifying Data Types in IDL for C through Table 3.9 Specifying Data Types in IDL for Java. Use these tables to map primitive types to their equivalent IDL syntax, and vice versa.
For C and Traditional C++, *RTI Code Generator* uses typedefs instead of the language keywords for primitive types. For example, DDS_Long instead of long (or int32) or DDS_Double instead of double. This ensures that the types are of the same size regardless of the platform.

The number of bytes sent on the wire for each data type is determined by the Common Data Representation (CDR) defined in the [OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3](https://www.omg.org/spec/DDS/1.3). There are two different CDR representations, encoding version 1 and encoding version 2. *Connext DDS* implements both. See the [RTI Connext DDS Core Libraries Extensible Types Guide](https://www.rti.com) for more information.

### 3.3.1 Variable-Length Types

When *RTI Code Generator* generates code for data structures with variable-length types—strings and sequences—it includes functions that create, initialize and finalize (destroy) those objects. These support functions will properly initialize pointers and allocate and deallocate the memory used for variable-length types. All *Connext DDS* APIs assume that the data structures passed to them are properly initialized.

For variable-length types, the actual length (instead of the maximum length) of data is transmitted on the wire when the DDS sample is written (regardless of whether the type has hard-coded bounds).

#### 3.3.1.1 Sequences

In C, Traditional C++, C++/CLI, C#, and Ada, sequences provide the concept of memory "ownership." A sequence may own the memory allocated to it or be loaned memory from another source. If a sequence owns its memory, it will manage its underlying memory storage buffer itself. When a sequence's maximum size is changed, the sequence will free and reallocate its buffer as needed. However, if a sequence was created with loaned memory by user code, then its memory is not its own to free or reallocate. Therefore, you cannot set the maximum size of a sequence whose memory is loaned. See the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Infrastructure Module, Sequence Support) for more information about how to loan and unloan memory for sequence.

In IDL, as described above, a sequence may be declared as bounded or unbounded. A sequence's "bound" is the greatest value its maximum may take. If you use the initializer functions *RTI Code Generator* provides for your types, all sequences will have their maximums set to their declared bounds. However, the amount of data transmitted on the wire when the DDS sample is written will vary.

In the the Modern C++ and Java APIs, sequences always own the memory.

#### 3.3.1.2 Strings and Wide Strings

**Note:** This section doesn't apply to the Modern C++ API, where strings map to `std::string` or `dds::core::string`, which behaves similarly. It also does not apply to the Traditional C++ API when generating code with the option `-useStdString`, which maps strings to `std::string`.

The initialization functions that *RTI Code Generator* provides for your types will allocate all of the memory for strings in a type to their declared bounds. Take care—if you assign a string pointer (char *) in
3.3.2 Value Types

a data structure allocated or initialized by a Connext DDS-generated function, you should release (free) the memory originally allocated for the string, otherwise the memory will be leaked.

To Java and .NET users, an IDL string is a String object: it is immutable and knows its own length. C and C++ users must take care, however, as there is no way to determine how much memory is allocated to a character pointer "string"; all that can be determined is the string's current logical length. In some cases, Connext DDS may need to copy a string into a structure that user code has provided. Connext DDS does not free the memory of the string provided to it, as it cannot know from where that memory was allocated.

In the C and C++ APIs, Connext DDS therefore uses the following conventions:

- A string's memory is "owned" by the structure that contains that string. Calling the finalization function provided for a type will free all recursively contained strings. If you have allocated a contained string in a special way, you must be careful to clean up your own memory and assign the pointer to NULL before calling the type’s finalize() method, so that Connext DDS will skip over that string.
- You must provide a non-NULL string pointer for Connext DDS to copy into. Otherwise, Connext DDS will log an error.
- When you provide a non-NULL string pointer in your data structure, Connext DDS will copy into the provided memory without performing any additional memory allocations. Be careful—if you provide Connext DDS with an uninitialized pointer or allocate a string that is too short, you may corrupt the memory or cause a program crash. Connext DDS will never try to copy a string that is longer than the bound of the destination string. However, your application must insure that any string that it allocates is long enough.

Connext DDS provides a small set of C functions for dealing with strings. These functions simplify common tasks, avoid some platform-specific issues (such as the lack of a strdup() function on some platforms), and provide facilities for dealing with wide strings, for which no standard C library exists. Connext DDS always uses these functions internally for managing string memory; you are recommended—but not required—to use them as well. See the API Reference HTML documentation, which is available for all supported programming languages (select Modules, RTI DDS API Reference, Infrastructure Module, String Support) for more information about strings.

3.3.2 Value Types

With the addition of inheritance to structs in Connext DDS 5.0, value types are considered equivalent to structs. It is recommended to use structures instead of value types, since the valuetype construct maybe not be supported in future releases. For additional information, see Structure Inheritance, in the Type System Enhancements chapter of the RTI Connext DDS Core Libraries Extensible Types Guide.

Readers familiar with value types in the context of CORBA should consult Table 3.4 Value Type Support to see which value type-related IDL keywords are supported and what their behavior is in the context of Connext DDS.
### 3.3.3 Type Codes

Type codes are always enabled when you run **RTI Code Generator**. Locally, your application can access the type code for a generated type "Foo" by calling the `FooTypeSupport::get_typecode()` (Traditional C++ Notation) operation in the code for the type generated by **RTI Code Generator**.

### 3.3.4 Translations for IDL Types

This section describes how to specify your data types in an IDL file. **RTI Code Generator** supports all the types listed in the following tables:

- Table 3.5 Specifying Data Types in IDL for C
- Table 3.6 Specifying Data Types in IDL for Traditional C++
- Table 3.7 Specifying Data Types in IDL for C++/CLI
- Table 3.8 Specifying Data Types in IDL for the Modern C++ API
- Table 3.9 Specifying Data Types in IDL for Java
- Table 3.10 Specifying Data Types in IDL for Ada

In each table, the middle column shows the IDL syntax for a data type in an IDL file. The rightmost column shows the corresponding language mapping created by **RTI Code Generator**.

---

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Level of Support in RTI Code Generator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inheritance</td>
<td>Single inheritance from other value types</td>
</tr>
<tr>
<td>Public state members</td>
<td>Supported</td>
</tr>
<tr>
<td>Private state members</td>
<td>Become public when code is generated</td>
</tr>
<tr>
<td>Custom keyword</td>
<td>Ignored (the value type is parsed without the keyword and code is generated to work with it)</td>
</tr>
<tr>
<td>Abstract value types</td>
<td>No code generated (the value type is parsed, but no code is generated)</td>
</tr>
<tr>
<td>Operations</td>
<td>No code generated (the value type is parsed, but no code is generated)</td>
</tr>
<tr>
<td>Truncatable keyword</td>
<td>Ignored (the value type is parsed without the keyword and code is generated to work with it)</td>
</tr>
</tbody>
</table>
### Table 3.5 Specifying Data Types in IDL for C

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td><code>struct PrimitiveStruct { char char_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_Char char_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>wchar</td>
<td><code>struct PrimitiveStruct { wchar wchar_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_Wchar wchar_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>octet</td>
<td><code>struct PrimitiveStruct { octet octet_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_Octet octet_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>int8</td>
<td><code>struct PrimitiveStruct { int8 int8_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_Int8 int8_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>uint8</td>
<td><code>struct PrimitiveStruct { uint8 uint8_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_UInt8 uint8_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>int16 or short</td>
<td><code>struct PrimitiveStruct { int16 short_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_Short short_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>uint16 or unsigned short</td>
<td><code>struct PrimitiveStruct { uint16 unsigned_short_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDSUnsignedShort unsigned_short_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>int32 or long</td>
<td><code>struct PrimitiveStruct { int32 long_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_Long long_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>uint32 or unsigned long</td>
<td><code>struct PrimitiveStruct { uint32 unsigned_long_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDSUnsignedLong unsigned_long_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>int64 or long long</td>
<td><code>struct PrimitiveStruct { int64 long_long_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDSLongLong long_long_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>uint64 or unsigned long long</td>
<td><code>struct PrimitiveStruct { uint64 unsigned_long_long_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDSUnsignedLongLong unsigned_long_long_member; } PrimitiveStruct;</code></td>
</tr>
<tr>
<td>float</td>
<td><code>struct PrimitiveStruct { float float_member; }</code></td>
<td><code>typedef struct PrimitiveStruct { DDS_Float float_member; } PrimitiveStruct;</code></td>
</tr>
</tbody>
</table>
### Table 3.5 Specifying Data Types in IDL for C

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>struct PrimitiveStruct { double double_member; }</td>
<td>typedef struct PrimitiveStruct { DDS_Double double_member; } PrimitiveStruct;</td>
</tr>
<tr>
<td>long double</td>
<td>struct PrimitiveStruct { long double long_double_member; }</td>
<td>typedef struct PrimitiveStruct { DDS_LongDouble long_double_member; } PrimitiveStruct;</td>
</tr>
<tr>
<td>@external or pointer</td>
<td>struct MyStruct {  @external long member; } or struct MyStruct { long * member; }</td>
<td>typedef struct MyStruct { DDS_Long * member; } MyStruct;</td>
</tr>
<tr>
<td>boolean</td>
<td>struct PrimitiveStruct { boolean boolean_member; }</td>
<td>typedef struct PrimitiveStruct { DDS_Boolean boolean_member; } PrimitiveStruct;</td>
</tr>
<tr>
<td>enum</td>
<td>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 };</td>
<td>typedef enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 } PrimitiveEnum;</td>
</tr>
<tr>
<td></td>
<td>enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</td>
<td>typedef enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 } PrimitiveEnum;</td>
</tr>
<tr>
<td></td>
<td>enum PrimitiveEnum { @value(10) ENUM1, @value(20) ENUM2, @value(30) ENUM3 }</td>
<td>typedef enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 } PrimitiveEnum;</td>
</tr>
<tr>
<td>constant</td>
<td>const short SIZE = 5;</td>
<td>#define SIZE 5</td>
</tr>
<tr>
<td>struct</td>
<td>struct PrimitiveStruct { char char_member; }</td>
<td>typedef struct PrimitiveStruct { char char_member; } PrimitiveStruct;</td>
</tr>
<tr>
<td>(see Note 10 below)</td>
<td>struct MyBaseStruct { long member_1; }</td>
<td>typedef struct MyBaseStruct { DDS_Long member_1; } MyBaseStruct;</td>
</tr>
<tr>
<td>struct inheritance</td>
<td>struct MyStruct: MyBaseStruct { long member_2; }</td>
<td>typedef struct MyStruct { MyBaseStruct parent; DDS_Long member_2; } MyStruct;</td>
</tr>
</tbody>
</table>

**Notes:**
- **Note 2:** long double is used for double precision floating-point numbers.
- **Note 9:** @external or pointer can be used to specify external memory.
- **Note 10:** struct can have members of other structures.

---

81
### Table 3.5 Specifying Data Types in IDL for C

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>union PrimitiveUnion&lt;br&gt;switch (long){&lt;br&gt;case 1:&lt;br&gt;    short short_member;&lt;br&gt;    default:&lt;br&gt;    long long_member;&lt;br&gt;}&lt;br&gt;</td>
<td>typedef struct PrimitiveUnion&lt;br&gt;    {&lt;br&gt;    DDS_Long _d;&lt;br&gt;    struct {&lt;br&gt;    DDS_Short short_member;&lt;br&gt;    DDS_Long long_member;&lt;br&gt;} _u;&lt;br&gt;} PrimitiveUnion;</td>
</tr>
<tr>
<td>typedef array of above types</td>
<td>typedef short TypedefShort;&lt;br&gt;struct OneArrayStruct {&lt;br&gt;    short short_array[2];&lt;br&gt;}&lt;br&gt;struct TwoArrayStruct {&lt;br&gt;    short short_array[1][2];&lt;br&gt;}&lt;br&gt;</td>
<td>typedef struct OneArrayStruct&lt;br&gt;    {&lt;br&gt;    DDS_Short short_array[2];&lt;br&gt;} OneArrayStruct;&lt;br&gt;typedef struct TwoArrayStruct&lt;br&gt;    {&lt;br&gt;    DDS_Short short_array[1][2];&lt;br&gt;} TwoArrayStruct;</td>
</tr>
<tr>
<td>bounded sequence of above types</td>
<td>struct SequenceStruct {&lt;br&gt;    sequence&lt;short,4&gt; short_sequence;&lt;br&gt;}&lt;br&gt;</td>
<td>typedef struct SequenceStruct&lt;br&gt;    {&lt;br&gt;    DDSShortSeq short_sequence;&lt;br&gt;} SequenceStruct;&lt;br&gt;Note: Sequences of primitive types have been predefined by Connext DDS</td>
</tr>
<tr>
<td>unbounded sequence of above types</td>
<td>struct SequenceStruct {&lt;br&gt;    sequence&lt;short&gt; short_sequence;&lt;br&gt;}&lt;br&gt;</td>
<td>typedef struct SequenceStruct&lt;br&gt;    {&lt;br&gt;    DDSShortSeq short_sequence;&lt;br&gt;} SequenceStruct;&lt;br&gt;See Note 12 below.</td>
</tr>
<tr>
<td>array of sequences</td>
<td>struct ArraysOfSequences{&lt;br&gt;    sequence&lt;short,4&gt; sequences_array[2];&lt;br&gt;}&lt;br&gt;</td>
<td>typedef struct ArraysOfSequences&lt;br&gt;    {&lt;br&gt;    DDSShortSeq sequences_array[2];&lt;br&gt;} ArraysOfSequences;</td>
</tr>
<tr>
<td>sequence of arrays</td>
<td>typedef short ShortArray[2];&lt;br&gt;struct SequenceOfArrays {&lt;br&gt;    sequence&lt;ShortArray,2&gt; arrays_sequence;&lt;br&gt;}&lt;br&gt;</td>
<td>typedef DDS_Short ShortArray[2];&lt;br&gt;DDS_SEQUENCE_NO_GET(ShortArraySeq,ShortArray);&lt;br&gt;typedef struct SequenceOfArrays&lt;br&gt;    {&lt;br&gt;    ShortArraySeq arrays_sequence;&lt;br&gt;} SequenceOfArrays;&lt;br&gt;DDS_SEQUENCE_NO_GET is a Connext DDS macro that defines a new sequence type for a user data type. In this case, the user data type is ShortArray.</td>
</tr>
</tbody>
</table>
### Table 3.5 Specifying Data Types in IDL for C

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| sequence of sequences     | typedef sequence<short,4> ShortSequence;  
                        | struct SequencesOfSequences {  
                        | sequence<ShortSequence,2> sequences_sequence;  
                        | };  
                        | typedef DDS_LongSeq ShortSequence;  
                        | DDS_SEQUENCE(ShortSequenceSeq,  
                        | ShortSequence);  
                        | typedef struct  
                        | SequencesOfSequences {  
                        | ShortSequenceSeq sequences_sequence;  
                        | } SequencesOfSequences; |
| bounded string            | struct PrimitiveStruct {  
                        | string<20> string_member;  
                        | };  
                        | typedef struct PrimitiveStruct {  
                        | DDS_Char* string_member;  
                        | /* maximum length = (20) */  
                        | } PrimitiveStruct; |
| unbounded string          | struct PrimitiveStruct {  
                        | string string_member;  
                        | };  
                        | typedef struct PrimitiveStruct {  
                        | DDS_Char* string_member;  
                        | /* maximum length = (255) */  
                        | } PrimitiveStruct;  
                        | See Note 12 below. |
| bounded wstring           | struct PrimitiveStruct {  
                        | wstring<20> wstring_member;  
                        | };  
                        | typedef struct PrimitiveStruct {  
                        | DDS_Wchar* wstring_member;  
                        | /* maximum length = (20) */  
                        | } PrimitiveStruct; |
| unbounded wstring         | struct PrimitiveStruct {  
                        | wstring wstring_member;  
                        | };  
                        | typedef struct PrimitiveStruct {  
                        | DDS_Wchar* wstring_member;  
                        | /* maximum length = (255) */  
                        | } PrimitiveStruct;  
                        | See Note 12 below. |
| module                    | module PackageName {  
                        | struct Foo {  
                        | long field;  
                        | } Foo;  
                        | };  
                        | With the -namespace option (only available for C++):  
                        | namespace PackageName{  
                        | typedef struct Foo {  
                        | DDS_Long field;  
                        | } Foo;  
                        | };  
                        | Without the -namespace option:  
                        | typedef struct PackageName_Foo {  
                        | DDS_Long field;  
                        | } PackageName_Foo;  
| valuetype                 | valuetype MyValueType {  
                        | public MyValueType2 * member;  
                        | };  
                        | typedef struct MyValueType {  
                        | MyValueType2 * member;  
                        | } MyValueType;  
                        | valuetype MyValueType {  
                        | public MyValueType2 member;  
                        | };  
                        | typedef struct MyValueType {  
                        | MyValueType2 member;  
                        | } MyValueType;  
                        | valuetype MyValueType;  
                        | MyBaseValueType {  
                        | public MyValueType2 * member;  
                        | };

(see Note 9 and Note 10 below)
### Table 3.6 Specifying Data Types in IDL for Traditional C++

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| char (see Note 1 below) | struct PrimitiveStruct { 
|                  |     char char_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS_Char char_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| wchar            | struct PrimitiveStruct { 
|                  |     wchar wchar_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS_Wchar wchar_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| octet            | struct PrimitiveStruct { 
|                  |     octet octet_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS_Octet octet_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| int8 (see Note 16 below) | struct PrimitiveStruct { 
|                  |     int8 int8_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS_Int8 int8_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| uint8 (see Note 16 below) | struct PrimitiveStruct { 
|                  |     uint8 uint8_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS(UInt8 uint8_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| int16 or short   | struct PrimitiveStruct { 
|                  |     int16 short_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS_Short short_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| uint16 or unsigned short | struct PrimitiveStruct { 
|                  |     uint16 unsigned_short_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDSUnsignedShort unsigned_short_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| int32 or long    | struct PrimitiveStruct { 
|                  |     int32 long_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS_Long long_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| uint32 or unsigned long | struct PrimitiveStruct { 
|                  |     uint32 unsigned_long_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDSUnsignedLong unsigned_long_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| int64 or long    | struct PrimitiveStruct { 
|                  |     int64 long_long_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDS_LongLong long_long_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| uint64 or unsigned long | struct PrimitiveStruct { 
|                  |     uint64 unsigned_long_long_member; 
|                  | };                                                                                     | class PrimitiveStruct 
|                  |     DDSUnsignedLongLong unsigned_long_long_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
| float            | struct PrimitiveStruct { 
|                  |     float float_member; 
|                  | };                                                                                     | typedef struct PrimitiveStruct 
|                  |     DDS_Float float_member; 
|                  | } PrimitiveStruct;                                                                      |                                                             |
### Table 3.6 Specifying Data Types in IDL for Traditional C++

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct {</td>
</tr>
<tr>
<td></td>
<td>double double_member;</td>
<td>DDS_Double double_member;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>} PrimitiveStruct;</td>
</tr>
<tr>
<td>long double</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct {</td>
</tr>
<tr>
<td>(see Note 2)</td>
<td>long double</td>
<td>DDS_LongDouble</td>
</tr>
<tr>
<td></td>
<td>long_double_member;</td>
<td>long_double_member;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>} PrimitiveStruct;</td>
</tr>
<tr>
<td>@external</td>
<td>struct MyStruct {</td>
<td>class MyStruct {</td>
</tr>
<tr>
<td>or pointer</td>
<td>@external long member;</td>
<td>DDS_Long * member;</td>
</tr>
<tr>
<td>(see Note 9)</td>
<td>or structure MyStruct {</td>
<td>} MyStruct;</td>
</tr>
<tr>
<td></td>
<td>long * member;</td>
<td></td>
</tr>
<tr>
<td>boolean</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct {</td>
</tr>
<tr>
<td></td>
<td>boolean boolean_member;</td>
<td>DDS_Boolean boolean_member;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>} PrimitiveStruct;</td>
</tr>
<tr>
<td>enum</td>
<td>enum PrimitiveEnum {</td>
<td>typedef enum PrimitiveEnum {</td>
</tr>
<tr>
<td></td>
<td>ENUM1,</td>
<td>ENUM1,</td>
</tr>
<tr>
<td></td>
<td>ENUM2,</td>
<td>ENUM2,</td>
</tr>
<tr>
<td></td>
<td>ENUM3</td>
<td>ENUM3</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>} PrimitiveEnum;</td>
</tr>
<tr>
<td></td>
<td>enum PrimitiveEnum {</td>
<td>typedef enum PrimitiveEnum {</td>
</tr>
<tr>
<td></td>
<td>ENUM1 = 10,</td>
<td>ENUM1 = 10,</td>
</tr>
<tr>
<td></td>
<td>ENUM2 = 20,</td>
<td>ENUM2 = 20,</td>
</tr>
<tr>
<td></td>
<td>ENUM3 = 30</td>
<td>ENUM3 = 30</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>} PrimitiveEnum;</td>
</tr>
<tr>
<td></td>
<td>enum PrimitiveEnum {</td>
<td>typedef enum PrimitiveEnum {</td>
</tr>
<tr>
<td></td>
<td>@value (10) ENUM1,</td>
<td>ENUM1 = 10,</td>
</tr>
<tr>
<td></td>
<td>@value (20) ENUM2,</td>
<td>ENUM2 = 20,</td>
</tr>
<tr>
<td></td>
<td>@value (30) ENUM3</td>
<td>ENUM3 = 30</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>} PrimitiveEnum;</td>
</tr>
<tr>
<td>constant</td>
<td>const short SIZE = 5;</td>
<td>static const DDS_Short size = 5;</td>
</tr>
<tr>
<td>struct</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct {</td>
</tr>
<tr>
<td>(see Note 10)</td>
<td>char char_member;</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>DDS_Char char_member;</td>
</tr>
<tr>
<td>struct</td>
<td>struct MyBaseStruct {</td>
<td>class MyBaseStruct {</td>
</tr>
<tr>
<td>inheritance</td>
<td>long member_1;</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>DDS_Long member_1;</td>
</tr>
<tr>
<td></td>
<td>struct MyStruct : MyBaseStruct {</td>
<td>class MyStruct : public MyBaseStruct {</td>
</tr>
<tr>
<td></td>
<td>long member_2;</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>DDS_Long member_2;</td>
</tr>
</tbody>
</table>
### Table 3.6 Specifying Data Types in IDL for Traditional C++

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td>union PrimitiveUnion switch (long) {</td>
<td>class PrimitiveUnion</td>
</tr>
<tr>
<td>(see Note 3 and Note 10 below)</td>
<td>case 1: short short_member; default: long long_member; }</td>
<td>class LONG {}</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>DDS_Long _d;</td>
</tr>
<tr>
<td></td>
<td>} PrimitiveUnion;</td>
<td>class {}</td>
</tr>
<tr>
<td></td>
<td>typedef short TypedefShort;</td>
<td>typedef DDS_Short TypedefShort;</td>
</tr>
<tr>
<td>array of above types</td>
<td>struct OneDArrayStruct { short short_array[2]; }</td>
<td>class OneDArrayStruct</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>DDS_Short short_array[2];</td>
</tr>
<tr>
<td></td>
<td>struct TwoDArrayStruct { short short_array[1][2]; }</td>
<td>} OneDArrayStruct;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>class TwoDArrayStruct</td>
</tr>
<tr>
<td>bounded sequence of above types</td>
<td>struct SequenceStruct ( sequence&lt;short,4&gt; short_sequence; )</td>
<td>class SequenceStruct</td>
</tr>
<tr>
<td>(see Note 11 and Note 15 below)</td>
<td>};</td>
<td>DDS_ShortSeq short_sequence;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>DDS_ShortSeq short_sequence; SequenceStruct;</td>
</tr>
<tr>
<td></td>
<td>Note: Sequences of primitive types have been predefined by Connext DDS.</td>
<td>Should Note 12 below.</td>
</tr>
<tr>
<td>unbounded sequence of above types</td>
<td>struct SequenceStruct ( sequence&lt;short&gt; short_sequence; )</td>
<td>typedef struct SequenceStruct</td>
</tr>
<tr>
<td>(see Note 11 and Note 15 below)</td>
<td>};</td>
<td>DDS_ShortSeq short_sequence; SequenceStruct;</td>
</tr>
<tr>
<td></td>
<td>array of sequences</td>
<td>See Note 12 below.</td>
</tr>
<tr>
<td>sequence of arrays</td>
<td>struct ArraysOfSequences ( sequence&lt;short,4&gt; sequences_array[2]; )</td>
<td>typedef DDS_Short ShortArray[2];</td>
</tr>
<tr>
<td>(see Note 11 below)</td>
<td>};</td>
<td>DDS_SEQUENCE_NO_GET(ShortArraySeq, ShortArray);</td>
</tr>
<tr>
<td></td>
<td>typedef short ShortArray[2];</td>
<td>class SequenceOfArrays</td>
</tr>
<tr>
<td></td>
<td>struct SequenceofArrays { sequence&lt;ShortArray,2&gt; arrays_sequence; }</td>
<td>ArraysOfSequences</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>DDS_ShortSeq arrays_sequence; SequenceOfArrays;</td>
</tr>
<tr>
<td></td>
<td>DDS_SEQUENCE_NO_GET is a Connext DDS macro that defines a new sequence type for a user data type. In this case, the user data type is ShortArray.</td>
<td></td>
</tr>
</tbody>
</table>


### 3.3.4 Translations for IDL Types

#### Table 3.6 Specifying Data Types in IDL for Traditional C++

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>sequence of sequences</strong></td>
<td>typedef sequence&lt;short,4&gt; ShortSequence;</td>
<td>typedef DDS_ShortSeq ShortSequence;</td>
</tr>
<tr>
<td>(see Note 4 and Note 11 below)</td>
<td>struct SequencesOfSequences(</td>
<td>DDS_SEQUENCE(ShortSequenceSeq, ShortSequence);</td>
</tr>
<tr>
<td></td>
<td>sequence&lt;ShortSequence,2&gt; sequences_sequence;</td>
<td>class SequencesOfSequences(</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>ShortSequenceSeq sequences_sequence;</td>
</tr>
<tr>
<td></td>
<td>)</td>
<td>SequencesOfSequences;</td>
</tr>
<tr>
<td><strong>bounded string</strong></td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>string&lt;20&gt; string_member;</td>
<td>DDS_Char* string_member;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>/* maximum length = (20) */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>) PrimitiveStruct;</td>
</tr>
<tr>
<td><strong>unbounded string</strong></td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>string string_member;</td>
<td>DDS_Char* string_member;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>/* maximum length = (255) */</td>
</tr>
<tr>
<td></td>
<td>See Note 12 below.</td>
<td>) PrimitiveStruct;</td>
</tr>
<tr>
<td><strong>bounded wstring</strong></td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>wstring&lt;20&gt; wstring_member;</td>
<td>DDS_Wchar* wstring_member;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>/* maximum length = (20) */</td>
</tr>
<tr>
<td></td>
<td></td>
<td>) PrimitiveStruct;</td>
</tr>
<tr>
<td><strong>unbounded wstring</strong></td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>wstring wstring_member;</td>
<td>DDS_Wchar* wstring_member;</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>/* maximum length = (255) */</td>
</tr>
<tr>
<td></td>
<td>See Note 12 below.</td>
<td>) PrimitiveStruct;</td>
</tr>
<tr>
<td><strong>module</strong></td>
<td>module PackageName {</td>
<td>With the -namespace option (only available for C++):</td>
</tr>
<tr>
<td></td>
<td>struct Foo {</td>
<td>namespace PackageName{</td>
</tr>
<tr>
<td></td>
<td>long field;</td>
<td>typedef struct Foo {</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>DDS_Long field;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>Foo;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>} PackageName_Foo;</td>
</tr>
<tr>
<td><strong>valuetype</strong></td>
<td>valuetype MyValueType (</td>
<td>With the -namespace option (only available for C++):</td>
</tr>
<tr>
<td>(see Note 9 and Note 10 below)</td>
<td>public MyValueType2 * member;</td>
<td>namespace PackageName{</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>typedef struct Foo {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDS_Long field;</td>
</tr>
<tr>
<td></td>
<td>valuetype MyValueType {</td>
<td>Foo;</td>
</tr>
<tr>
<td></td>
<td>public MyValueType2 member;</td>
<td>} PackageName_Foo;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>} PackageName_Foo;</td>
</tr>
<tr>
<td></td>
<td>valuetype MyValueType;</td>
<td>} PackageName_Foo;</td>
</tr>
<tr>
<td></td>
<td>MyBaseValueType {</td>
<td>} PackageName_Foo;</td>
</tr>
<tr>
<td></td>
<td>public MyValueType2 * member;</td>
<td>} PackageName_Foo;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>} PackageName_Foo;</td>
</tr>
<tr>
<td>IDL Type</td>
<td>Example Entry in IDL File</td>
<td>Example Output Generated by RTI Code Generator (rtiddsgen)</td>
</tr>
<tr>
<td>------------------</td>
<td>----------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>char (see Note 1 below)</td>
<td>struct PrimitiveStruct { char char_member; }</td>
<td>public ref class PrimitiveStruct { System::Char char_member; }</td>
</tr>
<tr>
<td>wchar</td>
<td>struct PrimitiveStruct { wchar wchar_member; }</td>
<td>public ref class PrimitiveStruct { System::Char wchar_member; }</td>
</tr>
<tr>
<td>octet</td>
<td>struct PrimitiveStruct { octet octet_member; }</td>
<td>public ref class PrimitiveStruct { System::Byte octet_member; }</td>
</tr>
<tr>
<td>int16 or short</td>
<td>struct PrimitiveStruct { int16 short_member; }</td>
<td>public ref class PrimitiveStruct { System::Int16 short_member; }</td>
</tr>
<tr>
<td>uint16 or unsigned short</td>
<td>struct PrimitiveStruct { unsigned_short_member; }</td>
<td>public ref class PrimitiveStruct { System::UInt16 unsigned_short_member; }</td>
</tr>
<tr>
<td>int32 or long</td>
<td>struct PrimitiveStruct { int32 long_member; }</td>
<td>public ref class PrimitiveStruct { System::Int32 long_member; }</td>
</tr>
<tr>
<td>uint32 or unsigned long</td>
<td>struct PrimitiveStruct { unsigned_long_member; }</td>
<td>public ref class PrimitiveStruct { System::UInt32 unsigned_long_member; }</td>
</tr>
<tr>
<td>int64 or long long</td>
<td>struct PrimitiveStruct { int64 long_long_member; }</td>
<td>public ref class PrimitiveStruct { System::Int64 long_long_member; }</td>
</tr>
<tr>
<td>uint64 or unsigned long long</td>
<td>struct PrimitiveStruct { unsigned_long_long_member; }</td>
<td>public ref class PrimitiveStruct { System::UInt64 unsigned_long_long_member; }</td>
</tr>
<tr>
<td>float</td>
<td>struct PrimitiveStruct { float float_member; }</td>
<td>public ref class PrimitiveStruct { System::Single float_member; }</td>
</tr>
<tr>
<td>double</td>
<td>struct PrimitiveStruct { double double_member; }</td>
<td>public ref class PrimitiveStruct { System::Double double_member; }</td>
</tr>
<tr>
<td>long double (see Note 2 below)</td>
<td>struct PrimitiveStruct { long_double long_double_member; }</td>
<td>public ref class PrimitiveStruct { DDS::LongDouble long_double_member; }</td>
</tr>
<tr>
<td>boolean</td>
<td>struct PrimitiveStruct { boolean boolean_member; }</td>
<td>public ref class PrimitiveStruct { System::Boolean boolean_member; }</td>
</tr>
</tbody>
</table>

**Table 3.7 Specifying Data Types in IDL for C++/CLI**

*Note 1: The char type is not considered an IDL type but is used as an example.*

*Note 2: The long double type is not considered an IDL type but is used as an example.*
### Table 3.7 Specifying Data Types in IDL for C++/CLI

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>enum</strong></td>
<td><code>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 };</code></td>
<td><code>public enum class PrimitiveEnum : System::Int32 { ENUM1, ENUM2, ENUM3 };</code></td>
</tr>
<tr>
<td></td>
<td><code>enum PrimitiveEnum { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</code></td>
<td><code>public enum class PrimitiveEnum : System::Int32 { ENUM1 = 10, ENUM2 = 20, ENUM3 = 30 };</code></td>
</tr>
<tr>
<td></td>
<td><code>enum PrimitiveEnum { @value (10) ENUM1, @value (20) ENUM2, @value (30) ENUM3 }</code></td>
<td></td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td><code>const short SIZE = 5;</code></td>
<td><code>public ref class SIZE {</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>   public:</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>System::Int16 VALUE = 5;</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>};</code></td>
</tr>
<tr>
<td><strong>struct</strong></td>
<td><code>struct PrimitiveStruct { char char_member; };</code></td>
<td><code>public ref class PrimitiveStruct { System::Char char_member; };</code></td>
</tr>
<tr>
<td>(see Note 10 below)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>struct inheritance</strong></td>
<td><code>struct MyBaseStruct { long member_1; };</code></td>
<td><code>public ref struct MyBaseStruct { System::Int32 member_1; };</code></td>
</tr>
<tr>
<td></td>
<td><code>struct MyStruct: MyBaseStruct { long member_2; };</code></td>
<td><code>public ref struct MyStruct: public MyBaseStruct { System::Int32 member_2; };</code></td>
</tr>
<tr>
<td><strong>union</strong></td>
<td><code>union PrimitiveUnion switch (long) { case 1: short short_member; default: long long_member; }</code></td>
<td><code>public ref class PrimitiveUnion { System::Int32 _d;</code></td>
</tr>
<tr>
<td>(see Note 3 and Note 10 below)</td>
<td></td>
<td><code>    struct PrimitiveUnion_u { System::Int16 short_member; System::Int32 long_member; } _u;</code></td>
</tr>
<tr>
<td><strong>array of above types</strong></td>
<td><code>struct OneDArrayStruct { short short_array[2]; };</code></td>
<td><code>public ref class OneDArrayStruct { array&lt;System::Int16&gt;^ short_array; /*length == 2*/ };</code></td>
</tr>
<tr>
<td><strong>bounded sequence of above types</strong> (see Note 11 and Note 15 below)</td>
<td><code>struct SequenceStruct { sequence&lt;short,4&gt; short_sequence; };</code></td>
<td><code>public ref class SequenceStruct { ShortSeq^ short_sequence; /*max = 4*/ };</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Note: Sequences of primitive types have been predefined by Connext DDS</td>
</tr>
</tbody>
</table>
### Table 3.7 Specifying Data Types in IDL for C++/CLI

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| unbounded sequence of above types (see Note 11 and Note 15 below) | struct SequenceStruct { 
    sequence<short> short_sequence; 
} ; | public ref class SequenceStruct { 
    ShortSeq^ short_sequence; 
    /*max = <default bound>*/ 
} ; |
| array of sequences | struct ArraysOfSequences{ 
    sequence<short,4> sequences_array[2]; 
} ; | public ref class ArraysOfSequences { 
    array<DDS::ShortSeq^>^ sequences_array; 
    // maximum length = (2) 
} ; |
| bounded string | struct PrimitiveStruct { 
    string20 string_member; 
} ; | public ref class PrimitiveStruct { 
    System::String^ string_member; 
    // maximum length = (20) 
} ; |
| unbounded string | struct PrimitiveStruct { 
    string string_member; 
} ; | public ref class PrimitiveStruct { 
    System::String^ string_member; 
    // maximum length = (255) 
} ; |
| bounded wstring | struct PrimitiveStruct { 
    wstring20 wstring_member; 
} ; | public ref class PrimitiveStruct { 
    System::String^ wstring_member; 
    // maximum length = (20) 
} ; |
| unbounded wstring | struct PrimitiveStruct { 
    wstring wstring_member; 
} ; | public ref class PrimitiveStruct { 
    System::String^ wstring_member; 
    // maximum length = (255) 
} ; |
| module | module PackageName { 
    struct Foo { 
        long field; 
    } ; 
} ; | namespace PackageName { 
    public ref class Foo { 
        System::Int32 field; 
    } ; 
} ; |

### Table 3.8 Specifying Data Types in IDL for the Modern C++ API

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| char | struct PrimitiveStruct { 
    char char_member; 
} ; | class PrimitiveStruct { 
    public: 
    char char_member() const OMG_NOEXCEPT; 
    void char_member(char value); 
} ; |
| wchar | struct PrimitiveStruct { 
    wchar wchar_member; 
} ; | class PrimitiveStruct { 
    public: 
    wchar_t wchar_member() const OMG_NOEXCEPT; 
    void wchar_member(wchar_t value); 
} ; |
### Table 3.8 Specifying Data Types in IDL for the Modern C++ API

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>octet</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td></td>
<td>octet octet_member;</td>
<td>uint8_t octet_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void octet_member(uint8_t value);</td>
</tr>
<tr>
<td>int8 (see Note 16 bel-</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td>ow)</td>
<td>int8 int8_member;</td>
<td>int8_t int8_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void int8_member(int8_t value);</td>
</tr>
<tr>
<td>uint8 (see Note 16 be-</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td>low)</td>
<td>uint8 uint8_member;</td>
<td>uint8_t uint8_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void uint8_member(uint8_t value);</td>
</tr>
<tr>
<td>int16 or short</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td></td>
<td>int16 short_member;</td>
<td>int16_t short_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void short_member(int16_t value);</td>
</tr>
<tr>
<td>uint16 or unsigned sh-</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td>rted short</td>
<td>uint16 unsigned_short_member;</td>
<td>uint16_t unsigned_short_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void unsigned_short_member(uint16_t value);</td>
</tr>
<tr>
<td>int32 or long</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td></td>
<td>int32 long_member;</td>
<td>int32_t long_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void long_member(int32_t value);</td>
</tr>
<tr>
<td>uint32 or unsigned l-</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td>ong</td>
<td>uint32 unsigned_long_member;</td>
<td>uint32_t unsigned_long_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void unsigned_long_member(uint32_t value);</td>
</tr>
<tr>
<td>int64 or long long</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td></td>
<td>int64_long_member;</td>
<td>rti::core::int64_long_long_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void long_long_member(rti::core::int64 value);</td>
</tr>
<tr>
<td>uint64 or unsigned lo-</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td>ng long</td>
<td>uint64 unsigned_long_long_member;</td>
<td>rti::core::uint64 unsigned_long_long_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td></td>
</tr>
<tr>
<td>float</td>
<td>struct PrimitiveStruct {</td>
<td>class PrimitiveStruct { public:</td>
</tr>
<tr>
<td></td>
<td>float float_member;</td>
<td>float float_member() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td>);</td>
<td>void float_member(float value);</td>
</tr>
</tbody>
</table>
### Table 3.8 Specifying Data Types in IDL for the Modern C++ API

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| double   | `struct PrimitiveStruct {  
double double_member;  
};` | `class PrimitiveStruct {  
public:  
double double_member() const OMG_NOEXCEPT;  
void double_member(double value);  
};` |
| long double (see Note 2 below) | `struct PrimitiveStruct {  
long double long_double_member;  
};` | `class PrimitiveStruct {  
public:  
rti::core::LongDouble& long_double_member()  
const OMG_NOEXCEPT;  
const rti::core::LongDouble& long_double_member()  
const OMG_NOEXCEPT;  
void long_double_member(  
const rti::core::LongDouble& value);  
};` |
| pointer (see Note 9 below) | `struct MyStruct {  
long * member;  
};` | `class PrimitiveStruct {  
public:  
dds::core::external<int32_t>& member();  
const dds::core::external<int32_t>& member() const;  
void member(dds::core::external<int32_t> value);  
};` |
| boolean | `struct PrimitiveStruct {  
boolean boolean_member;  
};` | `class PrimitiveStruct {  
public:  
bool boolean_member() const OMG_NOEXCEPT;  
void boolean_member(bool value);  
};` |
| enum | `enum PrimitiveEnum {  
ENUM1,  
ENUM2,  
ENUM3  
};
enum PrimitiveEnum {  
ENUM1 = 10,  
ENUM2 = 20,  
ENUM3 = 30  
};
enum PrimitiveEnum {  
@value (10) ENUM1,  
@value (20) ENUM2,  
@value (30) ENUM3  
};` | `enum class PrimitiveEnum {  
ENUM1,  
ENUM2,  
ENUM3  
};
enum class PrimitiveEnum {  
ENUM1 = 10,  
ENUM2 = 20,  
ENUM3 = 30  
};
enum class PrimitiveEnum {  
ENUM1 = 10,  
ENUM2 = 20,  
ENUM3 = 30  
};` |
| constant | `const short SIZE = 5;` | `static const int16_t SIZE = 5;` |
| struct (see Note 10 and Note 14 below) | `struct PrimitiveStruct {  
char char_member;  
};` | `class PrimitiveStruct {  
public:  
....  
char char_member() const OMG_NOEXCEPT;  
void char_member(char value);  
};` |
### 3.3.4 Translations for IDL Types

#### Table 3.8 Specifying Data Types in IDL for the Modern C++ API

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct inheritance</td>
<td><code>struct MyBaseStruct {</code> <code>long member_1;</code> <code>};</code></td>
<td><code>class MyBaseStruct</code> <code>public:</code> <code>int32_t&amp; member_1();</code> <code>const int32_t&amp; member_1() const;</code> <code>void member_1(int32_t value);</code> <code>};</code></td>
</tr>
<tr>
<td></td>
<td><code>struct MyStruct: MyBaseStruct {</code> <code>long member_2;</code> <code>};</code></td>
<td><code>class MyStruct: public MyBaseStruct</code> <code>public:</code> <code>int32_t&amp; member_2();</code> <code>const int32_t&amp; member_2() const;</code> <code>void member_2(int32_t value);</code> <code>};</code></td>
</tr>
<tr>
<td>union</td>
<td><code>union PrimitiveUnion switch (long){</code> <code>case 1:</code> <code>short short_member;</code> <code>default:</code> <code>long long_member;</code> <code>};</code></td>
<td><code>class PrimitiveUnion</code> <code>public:</code> <code>int32_t _d() const;</code> <code>void _d(int32_t value);</code> <code>int16_t _short_member() const;</code> <code>void short_member(int16_t value);</code> <code>int32_t _long_member() const;</code> <code>void long_member(int32_t value);</code> <code>static int32_t default_discriminator();</code> <code>};</code></td>
</tr>
<tr>
<td></td>
<td>(see Note 3 and Note 10 below)</td>
<td>private:<code> </code>int32_t _m_d;<code> </code>struct Union {<code> </code>int16_t _m_short_member;<code> </code>int32_t _m_long_member;<code> </code>Union();<code> </code>Union(int16_t _short_member,<code> </code>int32_t _long_member);<code> </code>};<code> </code>Union _m_u;<code> </code>};`</td>
</tr>
<tr>
<td>typedef</td>
<td><code>typedef short TypedefShort;</code></td>
<td><code>typedef int16_t TypedefShort;</code> <code>struct TypedefShort_AliasTag {</code> <code>};</code></td>
</tr>
<tr>
<td>array of above types</td>
<td><code>struct OneArrayStruct {</code> <code>short short_array[2];</code> <code>};</code></td>
<td><code>class OneArrayStruct</code> <code>public:</code> <code>dds::core::array&lt;int16_t, 2&gt;&amp; short_array()</code> <code>OMG_NOEXCEPT;</code> <code>const dds::core::array&lt;int16_t, 2&gt;&amp; short_array()</code> <code>const OMG_NOEXCEPT;</code> <code>void short_array(const dds::core::array</code> <code>&lt;int16_t, 2&gt;&amp; value);</code> <code>};</code></td>
</tr>
<tr>
<td></td>
<td><code>struct TwoArrayStruct {</code> <code>short short_array[1][2];</code> <code>};</code></td>
<td><code>class TwoArrayStruct</code> <code>public:</code> <code>dds::core::array&lt;dds::core::array&lt;int16_t, 2&gt;, 1&gt;</code> <code>&amp; short_array()</code> <code>OMG_NOEXCEPT;</code> <code>const dds::core::array&lt;dds::core::array</code> <code>&lt;int16_t, 2&gt;, 1&gt;&amp; short_array()</code> <code>const OMG_NOEXCEPT;</code> <code>void short_array(</code> <code>const dds::core::array</code> <code>&lt;dds::core::array&lt;int16_t, 2&gt;, 1&gt;&amp; value);</code> <code>};</code></td>
</tr>
</tbody>
</table>

Note 3: [Explanation of Note 3]

Note 10: [Explanation of Note 10]
### 3.3.4 Translations for IDL Types

**Table 3.8 Specifying Data Types in IDL for the Modern C++ API**

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounded sequence of above types</td>
<td>struct SequenceStruct {</td>
<td>class SequenceStruct {</td>
</tr>
<tr>
<td></td>
<td>sequence&lt;short,4&gt; short_sequence;</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>rti::core::bounded_sequence&lt;int16_t, 4&gt; &amp; short_sequence() OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>const rti::core::bounded_sequence &lt;int16_t, 4&gt;&amp; short_sequence() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void short_sequence(const rti::core::bounded_sequence&lt;int16_t, 4&gt;&amp; value);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>};</td>
</tr>
<tr>
<td>unbounded sequence of above types</td>
<td>struct SequenceStruct {</td>
<td>With -unboundedSupport, -alwaysUseStdVector, or the annotation @use_vector (see 3.3.9.7 The @use_vector annotation on page 119):</td>
</tr>
<tr>
<td>(see Note 15 below)</td>
<td>sequence&lt;short&gt; short_sequence;</td>
<td>class SequenceStruct {</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>std::vector&lt;int16_t&gt; &amp; short_sequence() OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>const std::vector&lt;int16_t&gt; &amp; short_sequence() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void short_sequence(const std::vector&lt;int16_t&gt;&amp; value);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>};</td>
</tr>
<tr>
<td>array of sequences</td>
<td>struct ArraysOfSequences(</td>
<td>class ArraysOfSequences {</td>
</tr>
<tr>
<td></td>
<td>sequence&lt;short,4&gt; sequences_array[2];</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>dds::core::array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;rti::core::bounded_sequence&lt;int16_t, 4&gt;, 2&gt; &amp; sequences_array() OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>const dds::core::array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;rti::core::bounded_sequence&lt;int16_t, 4&gt;, 2&gt; &amp; sequences_array() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void sequences_array()</td>
</tr>
<tr>
<td></td>
<td></td>
<td>const dds::core::array</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;rti::core::bounded_sequence&lt;int16_t, 4&gt;, 2&gt; &amp; value);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>};</td>
</tr>
<tr>
<td>sequence of arrays</td>
<td>typedef short ShortArray[2];</td>
<td>typedef dds::core::array&lt;int16_t, 2&gt; ShortArray;</td>
</tr>
<tr>
<td>(see Note 15 below)</td>
<td>struct SequenceofArrays(</td>
<td>class SequenceofArrays {</td>
</tr>
<tr>
<td></td>
<td>sequence&lt;ShortArray,2&gt; arrays_sequence;</td>
<td>public:</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rti::core::bounded_sequence&lt;ShortArray, 2&gt; &amp; arrays_sequence() OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>const rti::core::bounded_sequence&lt;ShortArray, 2&gt; &amp; arrays_sequence() const OMG_NOEXCEPT;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void arrays_sequence(</td>
</tr>
<tr>
<td></td>
<td></td>
<td>const rti::core::bounded_sequence&lt;ShortArray, 2&gt; &amp; value);</td>
</tr>
<tr>
<td></td>
<td></td>
<td>};</td>
</tr>
</tbody>
</table>

See 3.3.4 Translations for IDL Types on page 79.
### Table 3.8 Specifying Data Types in IDL for the Modern C++ API

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| sequence of       | typedef sequence<short,4> ShortSequence;                                                  | class SequencesOfSequences { public: ...
| sequences         | struct SequencesOfSequences{ sequence<ShortSequence,2> sequences_sequence; }              |     rti::core::bounded_sequence<ShortSequence, 2>
| (see Note 4       |                                                                                          | & sequences_sequence() const OMG_NOEXCEPT;
| below)            |                                                                                          |     void sequences_sequence(const
|                   |                                                                                          | rti::core::bounded_sequence<ShortSequence, 2>
|                   |                                                                                          | & value);                                                     |
|                   |                                                                                          | See 3.3.4 Translations for IDL Types on page 79.                                                                     |
| bounded string    | struct PrimitiveStruct { string<20> string_member; }                                      | class PrimitiveStruct {
|                   |                                                                                          |     std::string& string_member() const OMG_NOEXCEPT;
|                   |                                                                                          |     const std::string& string_member() const OMG_NOEXCEPT;
|                   |                                                                                          |     void string_member(const std::string& value);}
|                   |                                                                                          | See 3.3.4 Translations for IDL Types on page 79.                                                                     |
| unbounded string  | struct PrimitiveStruct { string string_member; }                                          | See Note 12 below.                                                                                                          |
|                   |                                                                                          | See 3.3.4 Translations for IDL Types on page 79.                                                                     |
| bounded wstring   | struct PrimitiveStruct { wstring<20> wstring_member; }                                     | class PrimitiveStruct {
|                   |                                                                                          |     std::wstring& wstring_member() const OMG_NOEXCEPT;
|                   |                                                                                          |     const std::wstring& wstring_member()
|                   |                                                                                          |     const OMG_NOEXCEPT;
|                   |                                                                                          |     void wstring_member(const std::wstring& value);
|                   |                                                                                          | };
|                   |                                                                                          | See 3.3.4 Translations for IDL Types on page 79.                                                                     |
| unbounded wstring | struct PrimitiveStruct { wstring wstring_member; }                                        | See Note 12 below.                                                                                                          |
|                   |                                                                                          | See 3.3.4 Translations for IDL Types on page 79.                                                                     |
| module            | module PackageName { struct Foo { long field; } }                                        | namespace PackageName {
|                   |                                                                                          |     class Foo {
|                   |                                                                                          |     int32_t field() const OMG_NOEXCEPT;
|                   |                                                                                          |     void field(int32_t value);
|                   |                                                                                          | };
|                   |                                                                                          | };

Note: See 3.3.4 Translations for IDL Types on page 79.
### Table 3.8 Specifying Data Types in IDL for the Modern C++ API

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>valuetype</td>
<td>value-type MyBaseValueType { public long member; };</td>
<td>class MyBaseValueType { public: int32_t member() const OMG_NOEXCEPT; void member(int32_t value);</td>
</tr>
<tr>
<td></td>
<td>value-type MyValueType: MyBaseValue {</td>
<td>class MyValueType : public MyBaseValueType { public: int16_t * member2() const OMG_NOEXCEPT; void member2(int16_t * value);</td>
</tr>
<tr>
<td>(see Note 9 and Note 10 below)</td>
<td>public short * member2; };</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.9 Specifying Data Types in IDL for Java

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Java Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct {</td>
</tr>
<tr>
<td>(see Note 5 below)</td>
<td>char char_member;</td>
<td>{ public char char_member; ... }</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td></td>
</tr>
<tr>
<td>wchar</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct {</td>
</tr>
<tr>
<td>(see Note 5 below)</td>
<td>wchar wchar_member;</td>
<td>{ public char wchar_member; ... }</td>
</tr>
<tr>
<td></td>
<td>};</td>
<td></td>
</tr>
<tr>
<td>octet</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct {</td>
</tr>
<tr>
<td></td>
<td>octet octet_member;</td>
<td>{ public byte octet_member; ... }</td>
</tr>
<tr>
<td>int8</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct {</td>
</tr>
<tr>
<td>(see Note 16 below)</td>
<td>int8 int8_member;</td>
<td>{ public byte int8_member; ... }</td>
</tr>
<tr>
<td>uint8</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct {</td>
</tr>
<tr>
<td>(see Note 16 below)</td>
<td>uint8 uint8_member;</td>
<td>{ public byte uint8_member; ... }</td>
</tr>
<tr>
<td>int16 or short</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct {</td>
</tr>
<tr>
<td></td>
<td>int16 short_member;</td>
<td>{ public short short_member; ... }</td>
</tr>
</tbody>
</table>
### Table 3.9 Specifying Data Types in IDL for Java

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL file</th>
<th>Example Java Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint16 or unsigned short</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>uint16</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned_short_member;</td>
<td>public short unsigned_short_member;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>(see Note 6 below)</td>
<td></td>
</tr>
<tr>
<td>int32 or long</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>int32 long_member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public int long_member;</td>
</tr>
<tr>
<td></td>
<td>(see Note 6 below)</td>
<td>...</td>
</tr>
<tr>
<td>uint32 or unsigned long</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>uint32</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unsigned_long_member;</td>
<td>public int unsigned_long_member;</td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>...</td>
</tr>
<tr>
<td></td>
<td>(see Note 6 below)</td>
<td></td>
</tr>
<tr>
<td>int64 or long long</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>int64 long_long_member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public long long_long_member;</td>
</tr>
<tr>
<td></td>
<td>(see Note 7 below)</td>
<td>...</td>
</tr>
<tr>
<td>uint64 or unsigned long long</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>uint64 long_long_long_member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public int unsigned_long_long_long_member;</td>
</tr>
<tr>
<td></td>
<td>(see Note 7 below)</td>
<td>...</td>
</tr>
<tr>
<td>float</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>float float_member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public float float_member;</td>
</tr>
<tr>
<td></td>
<td>(see Note 9 below)</td>
<td>...</td>
</tr>
<tr>
<td>double</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>double double_member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public double double_member;</td>
</tr>
<tr>
<td></td>
<td>(see Note 7 below)</td>
<td>...</td>
</tr>
<tr>
<td>long double</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>long double long_double_member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public double long_double_member;</td>
</tr>
<tr>
<td></td>
<td>(see Note 7 below)</td>
<td>...</td>
</tr>
<tr>
<td>pointer</td>
<td>struct MyStruct {</td>
<td>public class MyStruct</td>
</tr>
<tr>
<td></td>
<td>long * member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public int member;</td>
</tr>
<tr>
<td></td>
<td>(see Note 9 below)</td>
<td>...</td>
</tr>
<tr>
<td>boolean</td>
<td>struct PrimitiveStruct {</td>
<td>public class PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>boolean boolean_member;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td>public boolean boolean_member;</td>
</tr>
</tbody>
</table>

---

Note 6: Unsigned short and unsigned long types are available in IDL versions 2 and later.

Note 7: Long double types are available in IDL versions 3 and later.

Note 9: Pointers are supported in IDL versions 4 and later.
### Table 3.9 Specifying Data Types in IDL for Java

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL file</th>
<th>Example Java Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>enum</strong></td>
<td><code>enum PrimitiveEnum { ENUM1, ENUM2, ENUM3 };</code></td>
<td><code>public class PrimitiveEnum extends Enum {</code>&lt;br&gt;<code>  public static PrimitiveEnum ENUM1 = new PrimitiveEnum (&quot;ENUM1&quot;, 0);</code>&lt;br&gt;<code>  public static PrimitiveEnum ENUM2 = new PrimitiveEnum (&quot;ENUM2&quot;, 1);</code>&lt;br&gt;<code>  public static PrimitiveEnum ENUM3 = new PrimitiveEnum (&quot;ENUM3&quot;, 2);</code>&lt;br&gt;<code>  public static PrimitiveEnum valueOf(int ordinal);</code>&lt;br&gt;<code>  ... }</code></td>
</tr>
<tr>
<td><strong>constant</strong></td>
<td><code>const short SIZE = 5;</code></td>
<td><code>public class SIZE {</code>&lt;br&gt;<code>  public static final short VALUE = 5;</code>&lt;br&gt;<code>  }</code></td>
</tr>
<tr>
<td><strong>struct</strong></td>
<td><code>struct PrimitiveStruct { char char_member; }</code></td>
<td><code>public class PrimitiveStruct {</code>&lt;br&gt;<code>  public char char_member;</code>&lt;br&gt;<code>  }</code></td>
</tr>
<tr>
<td><strong>struct inheritance</strong></td>
<td><code>struct MyBaseStruct { long member_1; };</code>&lt;br&gt;<code>struct MyStruct: MyBaseStruct { long member_2; };</code></td>
<td><code>public class MyBaseStruct {</code>&lt;br&gt;<code>  public int member_1;</code>&lt;br&gt;<code>  };</code>&lt;br&gt;<code>public class MyStruct extends MyBaseStruct {</code>&lt;br&gt;<code>  public int member_2;</code>&lt;br&gt;<code>  };</code></td>
</tr>
<tr>
<td><strong>union</strong></td>
<td><code>union PrimitiveUnion switch (long){ case 1: short short_member; default: long long_member; };</code></td>
<td><code>public class PrimitiveUnion {</code>&lt;br&gt;<code>  public int _d;</code>&lt;br&gt;<code>  public short short_member;</code>&lt;br&gt;<code>  public int long_member;</code>&lt;br&gt;<code>  ...</code>&lt;br&gt;<code>  };</code></td>
</tr>
<tr>
<td><strong>typedef of primitives, enums, strings</strong></td>
<td><code>typedef short ShortType;</code>&lt;br&gt;<code>struct PrimitiveStruct { ShortType short_member; };</code></td>
<td><code>/* typedefs are unwounded to original type when used */</code>&lt;br&gt;<code>public class PrimitiveStruct {</code>&lt;br&gt;<code>  public short short_member;</code>&lt;br&gt;<code>  ...</code>&lt;br&gt;<code>  };</code></td>
</tr>
</tbody>
</table>
### Table 3.9 Specifying Data Types in IDL for Java

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL file</th>
<th>Example Java Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| typedef of sequences or arrays (see Note 8 below) | typedef short ShortArray[2]; | /* Wrapper class */
| | | public class ShortArray
| | | {  public short[] userData =
| | | new short[2];
| | |  ...  
| | |} |
| array | struct OneDArrayStruct {
| | short short_array[2];
| | } |
| | public class OneDArrayStruct
| | {  public short[] short_array =
| | new short[2];
| | ...  
| |} |
| bounded sequence (see Note 11 and Note 15 below) | struct TwoDArrayStruct {
| | short short_array[1][2];
| | } |
| | public class TwoDArrayStruct
| | {  public short[][] short_array =
| | new short[1][2];
| | ...  
| |} |
| | | Note: Sequences of primitive types have been predefined by Connext DDS. |
| unbounded sequence (see Note 11 and Note 15 below) | struct SequenceStruct {
| | sequence<short,4> short_sequence;
| | } |
| | public class SequenceStruct
| | {  public ShortSeq short_sequence =
| | new ShortSeq((4));
| | ...  
| |} |
| | | See Note 12 below. |
| array of sequences | struct ArraysOfSequences{
| | sequence<short,4> sequences_array[2];
| | } |
| | public class ArraysOfSequences
| | {  public ShortSeq[] sequences_array =
| | new ShortSeq[2];
| | ...  
| |} |
### Table 3.9 Specifying Data Types in IDL for Java

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL file</th>
<th>Example Java Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| sequence of arrays   | `typedef short ShortArray[2];
struct SequenceOfArrays{
    sequence<ShortArray,2> arrays_sequence;
};`                                                                                     | /* Wrapper class */
  public class ShortArray
  {
    public short[] userData =
        new short[2];
    ...
  }
  /* Sequence of wrapper class objects */
  public final class SequenceOfArrays
  {
    public ShortArraySeq arrays_sequence
        = new ShortArraySeq((2));
    ...
  }
| (see Note 11 below) |                                                                                          |                                                                                                                                 |
| sequence of sequences| `typedef sequence<short,4> ShortSequence;
struct SequenceOfSequences{
    sequence<ShortSequence,2> sequences_sequence;
};`                                                                                   | /* Wrapper class */
  public class ShortSequence
  {
    public ShortSeq userData
        = new ShortSeq((4));
    ...
  }
  /* Sequence of wrapper class objects */
  public final class SequenceOfSequences
  {
    public ShortSequenceSeq sequences_sequence
        = new ShortSequenceSeq((2));
    ...
  }
| (see Note 4 and Note 11 below) |                                                                                          |                                                                                                                                 |
| bounded string       | `struct PrimitiveStruct {
    string<20> string_member;
};`                                                                                      | public class PrimitiveStruct
  {
    public String string_member
        = new String();
    /* maximum length = (20) */
    ...
  }
| unbounded string     | `struct PrimitiveStruct {
    string string_member;
};`                                                                                       | public class PrimitiveStruct
  {
    public String string_member
        = new String();
    /* maximum length = (255) */
    ...
  }
|                                                                                             | See Note 12 below.                                                                      |
### 3.3.4 Translations for IDL Types

#### Table 3.9 Specifying Data Types in IDL for Java

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL file</th>
<th>Example Java Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounded wstring</td>
<td><code>struct PrimitiveStruct { wstring&lt;20&gt; wstring_member; }</code></td>
<td><code>public class PrimitiveStruct { public String wstring_member = new String(); /* maximum length = (20) */ ... }</code></td>
</tr>
<tr>
<td>unbounded wstring</td>
<td><code>struct PrimitiveStruct { wstring wstring_member; }</code></td>
<td><code>public class PrimitiveStruct { public String wstring_member = new String(); /* maximum length = (255) */ ... }</code></td>
</tr>
<tr>
<td>module</td>
<td><code>module PackageName { struct Foo { long field; } ; }</code></td>
<td><code>package PackageName; public class Foo { public int field; ... }</code></td>
</tr>
<tr>
<td>valuetype</td>
<td><code>valuetype MyValueType { public MyValueType2 * member; }</code></td>
<td><code>public class MyValueType { public MyValueType2 member; ... }</code></td>
</tr>
<tr>
<td></td>
<td>(`see Note 9 and Note 10 below)</td>
<td><code>public class MyValueType { public MyValueType2 member; ... }</code></td>
</tr>
<tr>
<td></td>
<td><code>valuetype MyValueType: MyBaseValueType { public MyValueType2 * member; }</code></td>
<td><code>public class MyValueType extends MyBaseValueType { public MyValueType2 member; ... }</code></td>
</tr>
</tbody>
</table>

#### Table 3.10 Specifying Data Types in IDL for Ada

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td><code>struct PrimitiveStruct { char char_member; }</code></td>
<td><code>type PrimitiveStruct is record char char_member : aliased Standard.DDS.Char; end record;</code></td>
</tr>
<tr>
<td>(<code>see Note 13 below</code>)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See Note 12 below.
<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wchar</td>
<td>struct PrimitiveStruct { wchar wchar_member; }</td>
<td>type PrimitiveStruct is record wchar_member: aliased Standard.DDS.Wchar; end record;</td>
</tr>
<tr>
<td>octet</td>
<td>struct PrimitiveStruct { octet octet_member; }</td>
<td>type PrimitiveStruct is record octet_member: aliased Standard.DDS.Octet; end record;</td>
</tr>
<tr>
<td>int16 or short</td>
<td>struct PrimitiveStruct { int16 short_member; }</td>
<td>type PrimitiveStruct is record short_member: aliased Standard.DDS.Short; end record;</td>
</tr>
<tr>
<td>uint16 or unsigned short</td>
<td>struct PrimitiveStruct { uint16 unsigned_short_member; }</td>
<td>type PrimitiveStruct is record unsigned_short_member: aliased Standard.DDS.Unsigned_short; end record;</td>
</tr>
<tr>
<td>int32 or long</td>
<td>struct PrimitiveStruct { int32 long_member; }</td>
<td>type PrimitiveStruct is record long_member: aliased Standard.DDS.Long; end record;</td>
</tr>
<tr>
<td>uint32 or unsigned long</td>
<td>struct PrimitiveStruct { uint32 unsigned_long_member; }</td>
<td>type PrimitiveStruct is record unsigned_long_member: aliased Standard.DDS.Unsigned_long; end record;</td>
</tr>
<tr>
<td>int64 or long long</td>
<td>struct PrimitiveStruct { int64 long_long_member; }</td>
<td>type PrimitiveStruct is record long_long_member: aliased Standard.DDS.Long_long; end record;</td>
</tr>
<tr>
<td>uint64 or unsigned long long</td>
<td>struct PrimitiveStruct { uint64 unsigned_long_long_member; }</td>
<td>type PrimitiveStruct is record unsigned_long_long_member: aliased Standard.DDS.Unsigned_long_long; end record;</td>
</tr>
<tr>
<td>float</td>
<td>struct PrimitiveStruct { float float_member; }</td>
<td>type PrimitiveStruct is record float_member: aliased Standard.DDS.Float; end record;</td>
</tr>
<tr>
<td>double</td>
<td>struct PrimitiveStruct { double double_member; }</td>
<td>type PrimitiveStruct is record double_member: aliased Standard.DDS.Double; end record;</td>
</tr>
<tr>
<td>long double</td>
<td>struct PrimitiveStruct { long double long_double_member; }</td>
<td>type PrimitiveStruct is record long_double_member: aliased Standard.DDS.Long_double; end record;</td>
</tr>
<tr>
<td>@external or pointer</td>
<td>struct MyStruct { @external long member; }</td>
<td>type MyStruct is record member: access Standard.DDS.Long; end record;</td>
</tr>
<tr>
<td>(see Note 9 below)</td>
<td>or struct MyStruct { long * member; }</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IDL Type</td>
<td>Example Entry in IDL File</td>
<td>Example Output Generated by RTI Code Generator (rtiddsgen)</td>
</tr>
<tr>
<td>----------</td>
<td>--------------------------</td>
<td>----------------------------------------------------------</td>
</tr>
</tbody>
</table>
| boolean  | struct PrimitiveStruct {
            boolean boolean_member;
            }
|          | type PrimitiveStruct is record
            boolean_member: aliased Standard.DDS.Boolean;
            end record; |
| enum     | enum PrimitiveEnum {
            ENUM1, 
            ENUM2, 
            ENUM3 
          };
|          | type PrimitiveEnum is
          (ENUM1, ENUM2, ENUM3);
|          | enum PrimitiveEnum {
            ENUM1 = 10, 
            ENUM2 = 20, 
            ENUM3 = 30 
          };
|          | for PrimitiveEnum use
          (ENUM1 => 10, ENUM2 => 20, ENUM3 => 30); |
| constant | const short SIZE = 5; |
|          | SIZE: constant
            Standard.DDS.Short := 5; |
| struct   | struct PrimitiveStruct {
            char char_member;
            }
|          | type PrimitiveStruct is record
            char_member: aliased Standard.DDS.Char;
            end record; |
| (see Note 10 below) | | |
| struct inheritance | struct MyBaseStruct {
            long member_1;
            }
|          | type MyBaseStruct is record
            member_1: aliased Standard.DDS.Long;
            end record; |
|          | struct MyStruct: MyBaseStruct {
            long member_2;
            }
|          | type MyStruct is record
            parent: aliased MyType_IDL_File.MyBaseStruct;
            member_2: aliased Standard.DDS.Long;
            end record; |
| union    | union PrimitiveUnion switch (long){
          case 1:
            short short_member;
          default:
            long long_member;
          } |
| (see Note 3 and Note 10 below) | type PrimitiveUnion is record
            short_member: aliased Standard.DDS.Short;
            long_member: aliased Standard.DDS.Long;
            end record; |
| typedef  | typedef short TypedefShort; |
|          | type TypedefShort is new
            Standard.DDS.Short; |
### 3.3.4 Translations for IDL Types

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
</table>
| array of above types | `struct OneDArrayStruct { 
short short_array[2];
}` | `type OneDArrayStruct is record 
short_array : aliased 
Standard.DDS.Short_Array(1..2);
end record;` |
|            | `struct TwoDArrayStruct { 
short short_array[1][2];
}` | `type TwoDArrayStruct is record 
short_array : aliased 
TwoDArrayStruct_short_array_Array; 
end record;` |
| bounded sequence of above types | `struct SequenceStruct { 
sequence<short,4> short_sequence;
}` | `type SequenceStruct is record 
short_sequence : aliased 
Standard.DDS.Short_Seq.Sequence;
end record;` |
|            | (see Note 11 and Note 15 below) | `See Note 13 below.` |
| unbounded sequence of above types | `struct SequenceStruct { 
sequence<short> short_sequence;
}` | `type SequenceStruct is record 
short_sequence : aliased 
Standard.DDS.Short_Seq.Sequence;
end record;` |
|            | (see Note 11 and Note 15 below) | `See Note 13 below.` |
| array of sequences | `struct ArraysOfSequences{ 
sequence<short,4> sequences_array[2];
}` | `type ArraysOfSequences_sequences_array_Array 
is array (1..2) of aliased 
Standard.DDS.Short_Seq.Sequence;
type ArraysOfSequences is record 
sequences_array : aliased 
ArraysOfSequences_sequences_array_Array;
end record;` |
| sequence of arrays | `typedef short ShortArray[2]; 
struct SequencesOfArrays { 
sequence<ShortArray,2> arrays_sequence;
}` | `type ShortArray is array (1..2) of Standard.DDS.Short;
...` |
|            | (see Note 11 below) | `type SequencesOfArrays is record 
arrays_sequence : aliased 
ADA_IDL_File.ShortArray_Seq.Sequence; 
end record;` |
| Note: ADA_IDL_File.ShortArray_Seq.Sequence is an instantiation of Standard.DDS.Sequences_Generic for the user's data type |
| sequence of sequences | `typedef sequence<short,4> ShortSequence; 
struct SequencesOfSequences{ 
sequence<ShortSequence,2> sequences_sequence;
}` | `type ShortSequence is new Standard.DDS.Short_Seq.Sequence; 
...` |
|            | (see Note 4 and Note 11 below) | `type SequencesOfSequences is record 
sequences_sequence : aliased 
ADA_IDL_File.ShortSequence_Seq.Sequence; 
end record;` |
| Note: ADA_IDL_File.ShortSequence_Seq.Sequence is an instantiation of Standard.DDS.Sequences_Generic for the user's data type |
### Table 3.10 Specifying Data Types in IDL for Ada

<table>
<thead>
<tr>
<th>IDL Type</th>
<th>Example Entry in IDL File</th>
<th>Example Output Generated by RTI Code Generator (rtiddsgen)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounded string</td>
<td>struct PrimitiveStruct { string&lt;20&gt; string_member; }</td>
<td>type PrimitiveStruct is record string_member : aliased Standard.DDS.String; -- maximum length = (20) end record;</td>
</tr>
<tr>
<td>unbounded string</td>
<td>struct PrimitiveStruct { string string_member; }</td>
<td>type PrimitiveStruct is record string_member : aliased Standard.DDS.String; -- maximum length = (255) end record;</td>
</tr>
<tr>
<td>boundedwstring</td>
<td>struct PrimitiveStruct { wstring&lt;20&gt; wstring_member; }</td>
<td>type PrimitiveStruct is record wstring_member : aliased Standard.DDS.Wide_String; -- maximum length = (20) end record;</td>
</tr>
<tr>
<td>unboundedwstring</td>
<td>struct PrimitiveStruct { wstring wstring_member; }</td>
<td>type PrimitiveStruct is record wstring_member : aliased Standard.DDS.Wide_String; -- maximum length = (255) end record;</td>
</tr>
<tr>
<td>module</td>
<td>module PackageName { struct Foo { long field; } }</td>
<td>package PackageName is type Foo is record field : aliased Standard.DDS.Long; end record; end PackageName;</td>
</tr>
<tr>
<td>valuetype</td>
<td>valuetype MyBaseValueType { valuetype MyBaseValueType { public long member; } }</td>
<td>type MyBaseValueType is record member : aliased Standard.DDS.Long; end record;</td>
</tr>
<tr>
<td>(see Note 9 and Note 10 below)</td>
<td>valuetype MyValueType: MyBaseValueType { public short * member2; }</td>
<td>type MyValueType is record parent : ADA_IDL_File.MyBaseValueType; member2 : access Standard.DDS.Short; end record;</td>
</tr>
</tbody>
</table>

### Notes for the above tables:

**Note 1:** In C and C++, primitive types are not represented as native language types (e.g. long, char, etc.) but as custom types in the DDS namespace (DDS_Long, DDS_Char, etc.). These typedefs are used to ensure that a field’s size is the same across platforms.

**Note 2:** Some platforms do not support long double or have different sizes for that type than defined by IDL (16 bytes). On such platforms, DDS_LongDouble (as well as the unsigned version) is mapped to a character array that matches the expected size of that type by default.

If you are using a platform whose native mapping has exactly the expected size, you can instruct Connext DDS to use the native type instead. That is, if sizeof(long double) == 16, you can tell Connext DDS to map DDS_LongDouble to long double.
3.3.4 Translations for IDL Types

double by defining the following macro either in code or on the compile line:

-DRTI_CDR_SIZEOF_LONG_DOUBLE=16

**Note 3:** Unions in IDL are mapped to structs in C, C++ and records in ADA, so that Connext DDS will not have to dynamically allocate memory for unions containing variable-length fields such as strings or sequences. To be efficient, the entire struct (or class in C++/CLI) is not sent when the union is published. Instead, Connext DDS uses the discriminator field of the struct to decide what field in the struct is actually sent on the wire.

**Note 4:** So-called "anonymous sequences"—sequences of sequences in which the sequence element has no type name of its own—are not supported in IDL. For example, this is not supported:

```cpp
sequence<sequence<short,4>,4> MySequence;
```

Sequences of typedef'ed types, where the typedef is really a sequence, are supported.

For example, this is supported:

```cpp
typedef sequence<short,4> MyShortSequence;
sequence<MyShortSequence,4> MySequence;
```

**Note 5:** IDL wchar and char are mapped to Java char, 16-bit unsigned quantities representing Unicode characters as specified in the standard OMG IDL to Java mapping. In C++/CLI, char and wchar are mapped to System::Char.

**Note 6:** The unsigned version for integer types is mapped to its signed version as specified in the standard OMG IDL to Java mapping.

**Note 7:** There is no current support in Java for the IDL long double type. This type is mapped to double as specified in the standard OMG IDL to Java mapping.

**Note 8:** Java does not have a typedef construct, nor does C++/CLI. Typedefs for types that are neither arrays nor sequences (struct, unions, strings, wstrings, primitive types and enums) are "unwound" to their original type until a simple IDL type or user-defined IDL type (of the non-typedef variety) is encountered. For typedefs of sequences or arrays, RTI Code Generator will generate wrapper classes.

**Note 9:** See 3.3.9.4 The @external Annotation on page 115.

**Note 10:** In-line nested types are not supported inside structures, unions or valuetypes. For example, this is not supported:

```cpp
struct Outer {
```
3.3.4 Translations for IDL Types

```c
short outer_short;
struct Inner {
    char inner_char;
    short inner_short;
} outer_nested_inner;
```

**Note 11:** The sequence `<Type>Seq` is implicitly declared in the IDL file and therefore it cannot be declared explicitly by the user. For example, this is not supported:

```c
typedef sequence<Foo> FooSeq; //error
```

However, if *RTI Code Generator's* option, `-typeSequenceSuffix <Suffix>`, is used and the `<Suffix>` is not 'Seq', the sequence would be:

```c
typedef sequence<Foo> Foo<Suffix>; //no error
```

**Note 12:** *RTI Code Generator* will supply a default bound for sequences and strings. You can specify that bound with the `-sequenceSize` or `-stringSize` command-line option, respectively. See the *RTI Code Generator User's Manual*.

**Note 13:** In ADA, primitive types are not represented as native language types (e.g., Character, etc.) but as custom types in the DDS namespace (Standard.DDS.Long, Standard.DDS.Char, etc.). These typedefs are used to ensure that a field’s size is the same across platforms.

**Note 14:** Every type provides a default constructor, a copy constructor, a move constructor, a constructor with parameters to set all the type's members, a destructor, a copy-assignment operator, and a move-assignment operator. Types also include equality operators, the operator `<<` and a namespace-level swap function.

```c
PrimitiveStruct();
explicit PrimitiveStruct(char char_member);
PrimitiveStruct(PrimitiveStruct&& other_) OMG_NOEXCEPT;
PrimitiveStruct& operator=(PrimitiveStruct&& other_) OMG_NOEXCEPT;
bool operator == (const PrimitiveStruct& other_) const;
bool operator != (const PrimitiveStruct& other_) const;
void swap(PrimitiveStruct& other_) OMG_NOEXCEPT;
std::ostream& operator << (std::ostream& o, const PrimitiveStruct& sample);
```

**Note 15:** Sequences of pointers are not supported. For example, this is NOT supported:

```c
sequence<long*, 100>;
```

Sequences of typedef'd types, where the typedef is really a pointer, are supported. For example, this is supported:

```c
typedef long* pointerToLong;
sequence<pointerToLong, 100>;
```
3.3.5 Escaped Identifiers

To use an IDL keyword as an identifier, the keyword must be “escaped” by prepending an underscore, ‘_’. In addition, you must run RTI Code Generator with the -enableEscapeChar option. For example:

```plaintext
struct MyStruct {
    octet _octet; // octet is a keyword. To use the type
    // as a member name we add '"'
};
```

The use of ‘_’ is a purely lexical convention that turns off keyword checking. The generated code will not contain ‘_’. For example, the mapping to C would be as follows:

```plaintext
struct MyStruct {
    unsigned char octet;
};
```

**Note:** If you generate code from an IDL file to a language ‘X’ (for example, C++), the keywords of this language cannot be used as IDL identifiers, even if they are escaped. For example:

```plaintext
struct MyStruct {
    int32 int; // error
    int32 _int; // error
};
```

3.3.6 Namespaces in IDL Files

In IDL, the module keyword is used to create namespaces for the declaration of types defined within the file.

Here is an example IDL definition:

```plaintext
module PackageName {
    struct Foo {
        int32 field;
    }
};
```

**C Mapping:**

The name of the module is concatenated to the name of the structure to create the namespace. The resulting code looks like this:

```plaintext
typedef struct PackageName_Foo {
    DDS_Long field;
} PackageName_Foo;
```

**C++ Mapping:**

In the Traditional C++ API, when using the -namespace command-line option, RTI Code Generator
generates a namespace, such as the following:

```cpp	namespace PackageName{
  class Foo {
    public:
      DDS_Long field;
  }
}
```

Without the `-namespace` option, the mapping adds the module to the name of the class:

```cpp
class PackageName_Foo {
  public:
    DDS_Long field;
}
```

In the Modern C++ API, namespaces are always used.

**C++/CLI Mapping:**

Independently of the usage of the `-namespace` command-line option, RTI Code Generator generates a namespace, such as the following:

```cpp	namespace PackageName{
  public ref struct Foo: public DDS::ICopyable<Foo^> { 
    public:
      System::Int32 field;
  };
}
```

**Java Mapping:**

A `Foo.java` file will be created in a directory called `PackageName` to use the equivalent concept as defined by Java. The file `PackageName/Foo.java` will contain a declaration of Foo class:

```java
package PackageName;
  public class Foo {
    public int field;
  };
```

In a more complex example, consider the following IDL definition:

```idl
module PackageName {
  struct Bar {
    int32 field;
  };
  struct Foo {
    Bar barField;
  };
}
```

When RTI Code Generator generates code for the above definition, it will resolve the `Bar` type to be within the scope of the `PackageName` module and automatically generate fully qualified type names.
3.3.6 Namespaces In IDL Files

C Mapping:

```c
typedef struct PackageName_Bar {
    DDS_Long field;
} PackageName_Bar;
typedef struct PackageName_Foo {
    PackageName_Bar barField;
} PackageName_Foo;
```

C++ Mapping:

With -namespace:

```cpp
namespace PackageName {
    class Bar {
        public:
            DDS_Long field;
    };
    class Foo {
        public:
            PackageName::Bar barField;
    };
};
```

Without -namespace:

```cpp
class PackageName_Bar {
    public:
        DDS_Long field;
};
class PackageName_Foo {
    public:
        PackageName_Bar barField;
};
```

C++/CLI Mapping:

```cpp
namespace PackageName{
    public ref struct Bar: public DDS::ICopyable<Bar^> { 
        public:
            System::Int32 field;
    };
    public ref struct Foo: public DDS::ICopyable<Foo^> { 
        public:
            PackageName::Bar^ barField;
    };
};
```

Java Mapping:

`PackageName/Bar.java` and `PackageName/Foo.java` would be created with the following code, respectively:
3.3.7 Referring to Other IDL Files

IDL files may refer to other IDL files using a syntax borrowed from C, C++, and C++/CLI preprocessors. For example:

**Bar.idl**

```idl
struct Bar {
};
```

**Foo.idl**

```idl
#include "Bar.idl"
struct Foo {
    Bar m1;
};
```

The parsing of Foo in the above scenario will be successful, since Bar can be found in Bar.idl. (If Bar was not declared in Bar.idl, Code Generator would report an error indicating that the symbol could not be found.)

When Code Generator uses the default preprocessor, it will look for the included files, in this example, Bar.idl, in the following directories and in this order:

1. Path designated by the operating system for temporary files (e.g., /tmp/ in Linux).
2. Working directory where Code Generator was executed.
3. Directory or directories specified by the user using the -I command-line option (if any).
4. Directory where the input file is.
5. Default C++ preprocessor's include directories (cpp -v /dev/null -o /dev/null).

Please note that when invoking Code Generator and specifying Foo.idl as a parameter, only the data types defined in that file will be generated. If Foo.idl includes another file, such as Bar.idl, you would also need to invoke Code Generator using Bar.idl as a parameter.

If Code Generator encounters an #include statement and you are generating code for C, C++, or C++/CLI, Code Generator will assume that code has been generated for Bar.idl with corresponding header files, Bar.h and BarPlugin.h.
3.3.8 Preprocessor Directives

The generated code will automatically add these files where needed in the Foo generated code, in order to compile correctly:

```
#include "Bar.h"
#include "BarPlugin.h"
```

Because Java types do not refer to one another in the same way, it is not possible for Code Generator to automatically generate Java import statements based on an IDL `#include` statement. `#include` statements will not generate any specific code when Java code is generated. To add imports to your generated Java code, you should use the `@copy` directive (see 3.3.9.5 The @copy and Related Annotations on page 116).

3.3.8 Preprocessor Directives

RTI Code Generator supports the standard preprocessor directives defined by the IDL specification, such as `#if`, `#endif`, `#include`, and `#define`.

To support these directives, RTI Code Generator calls an external C preprocessor before parsing the IDL file. On Windows systems, the preprocessor is `cl.exe`. On other architectures, the preprocessor is `cpp`.

You can change the default preprocessor with the `--ppPath` option. If you do not want to run the preprocessor, use the `--ppDisable` option (see the RTI Code Generator User's Manual).

3.3.9 Using Built-in Annotations

RTI Code Generator supports the following built-in annotations, which can be used in your IDL File:

- Described in this document:
  - `@key` (3.3.9.1 The @key Annotation on the next page)
  - `@nested` (3.3.9.2 The @nested Annotation on page 114)
  - `@value` (3.3.9.3 The @value Annotation on page 115)
  - `@external` (3.3.9.4 The @external Annotation on page 115)

- Described in the RTI Connext DDS Core Libraries Extensible Types Guide:
  - `@extensibility`
  - `@id`
  - `@hashid`
  - `@autoid`
  - `@optional`
  - `@appendable`
  - `@mutable`
3.3.9 Using Builtin Annotations

- @final
- @default
- @default_literal
- @min
- @max
- @range
- @allowed_data_representation

These annotations are described two standard documents: Interface Definition Language (Version 4) and Extensible and Dynamic Topic Types for DDS (Version 1.2).

In addition, RTI provides the following RTI-specific annotations:

- @copy (3.3.9.5 The @copy and Related Annotations on page 116)
- @resolve_name (3.3.9.6 The @resolve_name Annotation on page 117)
- @use_vector (3.3.9.7 The @use_vector annotation on page 119)
- @top_level (Replaced by @nested. See 3.3.9.2 The @nested Annotation on the next page.)
- @transfer_mode (3.3.9.8 The @transfer_mode annotation on page 119)
- @language_binding (3.3.9.9 The @language_binding Annotation on page 120)

## 3.3.9.1 The @key Annotation

To declare a key for your data type, insert the @key annotation in the IDL file before one or more fields of the data type.

With each key, Connext DDS associates an internal 16-byte representation, called a `key-hash`.

If the maximum size of the serialized key is greater than 16 bytes, to generate the key-hash, Connext DDS computes the MD5 key-hash of the serialized key in network-byte order. Otherwise (if the maximum size of the serialized key is <= 16 bytes), the key-hash is the serialized key in network-byte order.

Only `struct` and `valuetype` definitions in IDL may have key fields. When RTI Code Generator encounters @key, it considers the annotated field in the enclosing structure or valuetype to be part of the key. Table 3.11 Example Keys shows some examples of keys.
Table 3.11 Example Keys

<table>
<thead>
<tr>
<th>Type</th>
<th>Key Fields</th>
</tr>
</thead>
</table>
| struct NoKey {  
  int32 member1;  
  int32 member2;  
} |  |
| struct SimpleKey {  
  @key int32 member1;  
  int32 member2;  
} | member1 |
| struct NestedNoKey {  
  SimpleKey member1;  
  int32 member2;  
} |  |
| struct NestedKey {  
  @key SimpleKey member1;  
  int32 member2;  
} | member1.member1 |
| struct NestedKey2 {  
  @key NoKey member1;  
  int32 member2;  
} | member1.member1  
| } | member1.member2 |
| valuetype BaseValueKey {  
  @key public int32 member1;  
} | member1 |
| valuetype DerivedValueKey : BaseValueKey {  
  @key public int32 member2;  
} | member1 |
| valuetype DerivedValue : BaseValueKey {  
  public int32 member2;  
} | member1 |
| struct ArrayKey {  
  @key int32 member1[3];  
} | member1[0]  
| } | member1[1]  
| } | member1[2] |

3.3.9.2 The @nested Annotation

By default, RTI Code Generator generates user-level type-specific methods for all structures/unions found in an IDL file. These methods include the methods used by DataWriters and DataReaders to send and receive data of a given type. General methods for writing and reading that take a void pointer are not offered by Connext DDS because they are not type safe. Instead, type-specific methods must be created to support a particular data type.

We use the term ‘top-level type’ to refer to the data type for which you intend to create a DCPS Topic that can be published or subscribed to. For top-level types, RTI Code Generator must create all of the type-specific methods previously described in addition to the code to serialize/deserialize those types. However, some of structures/unions defined in the IDL file are only embedded within higher-level structures and are not meant to be published or subscribed to individually. For non-top-level types, the DataWriters and
DataReaders methods to send or receive data of those types are superfluous and do not need to be created. Although the existence of these methods is not a problem in and of itself, code space can be saved if these methods are not generated in the first place.

You can mark non-top-level types in an IDL file with the annotation @nested to tell RTI Code Generator not to generate type-specific methods. Code will still be generated to serialize and deserialize those types, since they may be embedded in top-level types.

The top-level directive can also be used but with the opposite meaning. @top_level or //@top-level (true) indicates that the type is top level, therefore, @top_level (false) would be equivalent to @nested.

In this example, RTI Code Generator will generate DataWriter/DataReader code for TopLevelStruct only:

```cpp
@nested
struct EmbeddedStruct { 
    int16 member;
};
struct TopLevelStruct{ 
    EmbeddedStruct member;
};
```

3.3.9.3 The @value Annotation

The @value annotation can be used to set specific values to members of enumerations. For example:

```cpp
enum MyEnum { 
    @value (17) e17, 
    @value (2) e2, 
    @value (3) e3 
}
```

It is equivalent to:

```cpp
enum MyEnum { 
    e17 =17, 
    e2 = 2, 
    e3 = 3 
}
```

3.3.9.4 The @external Annotation

A member declared as external using the @external annotation (or the * modifier) within an aggregated type indicates that it is desirable for the implementation to store the member in storage external to the enclosing aggregated type object.

For example:
3.3.9 Using Builtin Annotations

```c
struct MyStruct {
    @external int32 member;
}
```

This is equivalent to the following structure, although the usage of the `@external` annotation is preferred because it is standard:

```c
struct MyStruct {
    int32 *member;
};
```

The `@external` annotation only has effect in C, C++, Modern C++, and Ada applications where the members will be mapped to references (pointers). In other languages, the annotation is ignored because the members are always mapped as references.

In Modern C++ the annotation maps to the type `dds::core::external<T>`, a type similar to `shared_ptr`.

### 3.3.9.5 The `@copy` and Related Annotations

To copy a line of text verbatim into the generated code files, use the `@copy` annotation in the IDL file. The `@copy` annotation can only be applied using the comment syntax (`//@`). The `@copy` annotation is particularly useful when you want your generated code to contain text that is valid in the target programming language but is not valid IDL. It is often used to add user comments or headers or preprocessor commands into the generated code.

```c
//@copy (// Modification History)
//@copy (// ------------------------)
//@copy (// 17Jul05aaa, Created.)
//@copy (// #include "MyTypes.h")
```

These variations allow you to use the same IDL file for multiple languages:

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>@copy-c</code></td>
<td>Copies code if the language is C or C++</td>
</tr>
<tr>
<td><code>@copy-cppcli</code></td>
<td>Copies code if the language is C++/CLI</td>
</tr>
<tr>
<td><code>@copy-java</code></td>
<td>Copies code if the language is Java.</td>
</tr>
<tr>
<td><code>@copy-ada</code></td>
<td>Copies code if the language is Ada.</td>
</tr>
</tbody>
</table>

For example, to add import statements to generated Java code:

```c
//@copy-java (import java.util.*;)
```

The above line would be ignored if the same IDL file was used to generate non-Java code.

In C, C++, and C++/CLI, the lines are copied into all of the `foo*.[h, c, cxx, cpp]` files generated from `foo.idl`. For Java, the lines are copied into all of the `*.java` files that were generated from the original “.idl” file. The lines will not be copied into any additional files that are generated using the `-example` command line option.
@copy-java-begin copies a line of text at the beginning of all the Java files generated for a type. The annotation only applies to the first type that is immediately below in the IDL file. A similar annotation for Ada files is also available, @copy-ada-begin.

If you want RTI Code Generator to copy lines only into the files that declare the data types—foo.h for C, C++, and C++/CLI, foo.java for Java—use the //@copy*declaration forms of this annotation.

Note that the first whitespace character to follow //@copy is considered a delimiter and will not be copied into generated files. All subsequent text found on the line, including any leading whitespaces will be copied.

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>//@copy-declaration</td>
<td>Copies the text into the file where the type is declared (type.h for C and C++, or type.java for Java)</td>
</tr>
<tr>
<td>//@copy-c-declaration</td>
<td>Same as //@copy-declaration, but for C and C++ code</td>
</tr>
<tr>
<td>//@copy-cppcli-declaration</td>
<td>Same as //@copy-declaration, but for C++/CLI code</td>
</tr>
<tr>
<td>//@copy-java-declaration</td>
<td>Same as //@copy-declaration, but for Java-only code</td>
</tr>
<tr>
<td>//@copy-ada-declaration</td>
<td>Same as //@copy-declaration, but for Ada-only code</td>
</tr>
<tr>
<td>//@copy-java-declaration-begin</td>
<td>Same as //@copy-java-declaration, but only copies the text into the file where the type is declared</td>
</tr>
<tr>
<td>//@copy-ada-declaration-begin</td>
<td>Same as //@copy-java-declaration-begin, but only for Ada-only code</td>
</tr>
</tbody>
</table>

### 3.3.9.6 The @resolve_name Annotation

By default, RTI Code Generator tries to resolve all the references to types and constants in an IDL file. For example:

```idl
module PackageName {
    struct Foo {
        Bar barField;
    };
};
```

The compilation of the previous IDL file will report an error like the following:

```
ERROR com.rti.ndds.nddsgen.Main Foo.idl line x:x member type 'Bar' not found
```

In most cases, this is the expected behavior. However, in some cases, you may want to skip the resolution step. For example, assume that the Bar type is defined in a separate IDL file and that you are running RTI Code Generator without an external preprocessor by using the command-line option -ppDisable (maybe because the preprocessor is not available in their host platform, see 3.3.8 Preprocessor Directives on page 112):

```
Bar.idl
```
3.3.9 Using Builtin Annotations

```c
module PackageName {
    struct Bar {
        int32 field;
    };
}
```

**Foo.idl**

```c
#include "Bar.idl"
module PackageName {
    struct Foo {
        Bar barField;
    };
}
```

In this case, compiling `Foo.idl` would generate the 'not found' error. However, Bar is defined in Bar.idl.

To specify that RTI Code Generator should not resolve a type reference, use the `//@resolve_name` annotation and set the value to false. For example:

```c
#include "Bar.idl"
module PackageName {
    struct Foo {
        @resolve_name(false) Bar barField;
    };
}
```

When this annotation is used, then for the field to which it applies, RTI Code Generator will assume that the type is an unkeyed 'structure' and it will use the type name unmodified in the generated code.

**Java mapping:**

```java
package PackageName;
public class Foo {
    public Bar barField = Bar.create();
}
```

**C++ mapping:**

```cpp
namespace PackageName {
    class Foo {
        public:
            Bar barField;
    };
}
```

It is up to you to include the correct header files (or if using Java, to import the correct packages) so that the compiler resolves the 'Bar' type correctly. If needed, this can be done using the copy directives (see 3.3.9.5 The @copy and Related Annotations on page 116).

When applied to an aggregated type in IDL, the annotation applies to all types within the type, including the base type if defined. For example:

```c
@resolve_name(false)
struct MyStructure: MyBaseStructure {
    Foo member1;
}
```
3.3.9.7 The @use_vector annotation

The @use_vector annotation can be used in Modern C++ to indicate that a bounded sequence should be mapped to std::vector; otherwise it will be mapped to rti:core::bounded_sequence.

For example:

```cpp
struct MyStruct {
    @use_vector sequence<int32, 10> my_bounded_seq;
}
```

As an alternative, you can use `rtiddsgen's -alwaysUseStdVector` option to indicate that all bounded sequences should be mapped to std::vector. Unbounded sequences always map to std::vector.

3.3.9.8 The @transfer_mode annotation

The @transfer_mode annotation can be used to indicate how to send a sample of the annotated type. There are two possible values for this annotation: SHMEM_REF and INBAND.

The annotation can be used only while generating code for C and C++ (Traditional and Modern) APIs. For other languages, the annotation is ignored.

@transfer_mode(SHMEM_REF) indicates that a sample can be sent as a shared memory reference instead of sending the serialized sample, when the DataReader(s) are on the same node as the DataWriter writing the sample. See 23.1.5 Zero Copy Transfer Over Shared Memory on page 962 for more information.

@transfer_mode(INBAND) indicates that a sample is always serialized and sent inband using the underlying transports. This is the default mode when the annotation is not present.

The use of @transfer_mode annotation without a parameter is not allowed and will generate an error during code generation.

It is sufficient to mark only the top-level types with the @transfer_mode annotation. In this example, a sample of type CameraImage can be sent as a shared memory reference, even though the included type Dimension is not explicitly annotated:

```cpp
struct Dimension {
    int32 height;
    int32 width;
};

@transfer_mode(SHMEM_REF)
struct CameraImage {
    int64 timestamp;
    Dimension dimension;
    octet data[8294400][4];
};
```
RTI Code Generator will return an error while parsing the IDL file if the following requirements are not met:

- All fixed and appendable types (described in RTI Connext DDS Core Libraries Extensible Types Guide) annotated with @transfer_mode(SHMEM_REF) should be fixed-size types. A fixed-size type is a type whose wire representation always has the same size. This includes primitive members, arrays of fixed-size types, and structs containing only members of fixed-size types. In the above example, the types CameraImage and Dimension should not contain variable-length members such as strings, sequences, and optional and external members.

- Mutable types annotated with @transfer_mode(SHMEM_REF) can contain variable-length members when the type is also annotated with FLAT_DATA language binding.

The @transfer_mode annotation can be applied to modules, structs, valuetypes, and unions. When applied to a module, all the types within the module inherit the language binding value specified in the module.

### 3.3.9.9 The @language_binding Annotation

The @language_binding annotation allows selecting the language binding for a type, either the plain language binding (default option when the annotation is not specified) or the RTI FlatData™ language binding.

PLAIN is the regular language binding that maps IDL types to their regular C or C++ representation as C structs or C++ classes.

FLAT_DATA is a special language binding in which the in-memory representation is the same as the wire representation. See 23.1.4 FlatData Language Binding on page 949 for a detailed description.

For example:

```idl
@language_binding(PLAIN) // or no annotation
struct MyNormalType {
    ...
};

@language_binding(FLAT_DATA)
struct MyFlatType {
    ...
};
```

A few notes about the @language_binding annotation:

- The annotation can be applied to modules, structs, valuetypes, and unions. When applied to a module, all the types within the module inherit the language binding value specified in the module.
3.4 Creating User Data Types with Extensible Markup Language (XML)

You can describe user data types with Extensible Markup Language (XML) notation. Connext DDS provides DTD and XSD files that describe the XML format; see `<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti.dds_topic_types.dtd` and `<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti.dds_topic_types.xsd`, respectively. (<NDDSHOME> is described in Paths Mentioned in Documentation on page 1.)

The XML validation performed by RTI Code Generator always uses the DTD definition. If the <!DOCTYPE> tag is not in the XML file, RTI Code Generator will look for the default DTD document in `<NDDSHOME>/resource/schema`. Otherwise, it will use the location specified in <!DOCTYPE>. We recommend including a reference to the XSD/DTD files in the XML documents. This provides helpful features in code editors such as Visual Studio® and Eclipse™, including validation and auto-completion while you are editing the XML. We recommend including the reference to the XSD document in the XML files because it provides stricter validation and better auto-completion than the DTD document.

To include a reference to the XSD document in your XML file, use the attribute xsi:noNamespaceSchemaLocation in the <types> tag. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti.dds_topic_types.xsd">
  ...
</types>
```

To include a reference to the DTD document in your XML file, use the <!DOCTYPE> tag. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE types SYSTEM "<NDDSHOME>/resource/app/app_support/rtiddsgen/schema/rti.dds_topic_types.dtd">
  <types>
    ...
  </types>
```

Table 3.12 Mapping Type System Constructs to XML shows how to map the type system constructs into XML. For information on the annotations in the table, see 3.3.9 Using Built-in Annotations on page 112.
### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XML</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>char8</td>
<td>char char_member;</td>
<td><code>&lt;member name=&quot;char_member&quot; type=&quot;char8&quot;/&gt;</code></td>
</tr>
<tr>
<td>wchar</td>
<td>char16</td>
<td>wchar wchar_member;</td>
<td><code>&lt;member name=&quot;wchar_member&quot; type=&quot;char16&quot;/&gt;</code></td>
</tr>
<tr>
<td>octet</td>
<td>byte</td>
<td>octet octet_member;</td>
<td><code>&lt;member name=&quot;octet_member&quot; type=&quot;byte&quot;/&gt;</code></td>
</tr>
<tr>
<td>int8¹</td>
<td>int8</td>
<td>int8 int8_member;</td>
<td><code>&lt;member name=&quot;int8_member&quot; type=&quot;int8&quot;/&gt;</code></td>
</tr>
<tr>
<td>uint8²</td>
<td>uint8</td>
<td>uint8 uint8_member;</td>
<td><code>&lt;member name=&quot;uint8_member&quot; type=&quot;uint8&quot;/&gt;</code></td>
</tr>
<tr>
<td>int16 or short</td>
<td>int16</td>
<td>int16 short_member;</td>
<td><code>&lt;member name=&quot;short_member&quot; type=&quot;int16&quot;/&gt;</code></td>
</tr>
<tr>
<td>uint16 or un-signed short</td>
<td>uint16</td>
<td>uint16 unsigned_short_member;</td>
<td><code>&lt;member name=&quot;unsigned_short_member&quot; type=&quot;uint16&quot;/&gt;</code></td>
</tr>
<tr>
<td>int32 or long</td>
<td>int32</td>
<td>int32 long_member;</td>
<td><code>&lt;member name=&quot;long_member&quot; type=&quot;int32&quot;/&gt;</code></td>
</tr>
<tr>
<td>uint32 or un-signed long</td>
<td>int32</td>
<td>uint32 unsigned_long_member;</td>
<td><code>&lt;member name=&quot;unsigned_long_member&quot; type=&quot;uint32&quot;/&gt;</code></td>
</tr>
<tr>
<td>int64 or long long</td>
<td>int64</td>
<td>int64 long_long_member;</td>
<td><code>&lt;member name=&quot;long_long_member&quot; type=&quot;int64&quot;/&gt;</code></td>
</tr>
<tr>
<td>uint64 or un-signed long long</td>
<td>int64</td>
<td>uint64 unsigned_long_long_member;</td>
<td><code>&lt;member name=&quot;unsigned_long_long_member&quot; type=&quot;uint64&quot;/&gt;</code></td>
</tr>
<tr>
<td>float</td>
<td>float32</td>
<td>float float_member;</td>
<td><code>&lt;member name=&quot;float_member&quot; type=&quot;float32&quot;/&gt;</code></td>
</tr>
<tr>
<td>double</td>
<td>float64</td>
<td>double double_member;</td>
<td><code>&lt;member name=&quot;double_member&quot; type=&quot;float64&quot;/&gt;</code></td>
</tr>
<tr>
<td>long double</td>
<td>float128</td>
<td>long double long_double_member;</td>
<td><code>&lt;member name=&quot;long_double_member&quot; type=&quot;float128&quot;/&gt;</code></td>
</tr>
</tbody>
</table>

¹This type is supported only at the API level. It is still considered an octet for type matching purposes.

²This type is supported only at the API level. It is still considered an octet for type matching purposes.
### 3.4 Creating User Data Types with Extensible Markup Language (XML)

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XML</strong></td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
</tbody>
</table>
| struct PrimitiveStruct {  
  boolean boolean_member;  
} | <struct name="PrimitiveStruct">  
  <member name="boolean_member"  
      type="boolean"/>  
</struct> |
| string         | string |
| struct PrimitiveStruct {  
  string string_member;  
} | <struct name="PrimitiveStruct">  
  <member name="string_member"  
      type="string"/>  
</struct> |
| wstring        | wstring |
| struct PrimitiveStruct {  
  wstring wstring_member;  
} | <struct name="PrimitiveStruct">  
  <member name="wstring_member"  
      type="wstring"/>  
</struct> |
| enum           | enum |
| enum PrimitiveEnum {  
  ENUM1,  
  ENUM2,  
  ENUM3  
} | <enum name="PrimitiveEnum">  
  <enumerator name="ENUM1"/>
  <enumerator name="ENUM2"/>
  <enumerator name="ENUM3"/>
</enum> |
| constant       | const |
| const double PI = 3.1415; | <const name="PI" type="double"  
    value="3.1415"/> |
### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XML</strong></td>
</tr>
<tr>
<td>struct PrimitiveStruct { short short_member; }</td>
<td>&lt;struct name=&quot;PrimitiveStruct&quot;&gt; &lt;member name=&quot;short_member&quot; type=&quot;short&quot;/&gt; &lt;/struct&gt;</td>
</tr>
<tr>
<td>union PrimitiveUnion switch (long) { case 1: short short_member; case 2: case 3: float float_member; default: long long_member; }</td>
<td>&lt;union name=&quot;PrimitiveUnion&quot;&gt; &lt;discriminator type=&quot;long&quot;/&gt; &lt;case&gt; &lt;caseDiscriminator value=&quot;1&quot;/&gt; &lt;member name=&quot;short_member&quot; type=&quot;short&quot;/&gt; &lt;/case&gt; &lt;case&gt; &lt;caseDiscriminator value=&quot;2&quot;/&gt; &lt;caseDiscriminator value=&quot;3&quot;/&gt; &lt;member name=&quot;float_member&quot; type=&quot;float&quot;/&gt; &lt;/case&gt; &lt;case&gt; &lt;caseDiscriminator value=&quot;default&quot;/&gt; &lt;member name=&quot;long_member&quot; type=&quot;long&quot;/&gt; &lt;/case&gt; &lt;/union&gt;</td>
</tr>
<tr>
<td>valuetype BaseValueType { public long long_member; }</td>
<td>&lt;valuetype name=&quot;BaseValueType&quot;&gt; &lt;member name=&quot;long_member&quot; type=&quot;long&quot; visibility=&quot;public&quot;/&gt; &lt;/valuetype&gt;</td>
</tr>
<tr>
<td>valuetype DerivedValueType: BaseValueType { public long long_member_2; }</td>
<td>&lt;valuetype name=&quot;DerivedValueType&quot; baseClass=&quot;BaseValueType&quot;&gt; &lt;member name=&quot;long_member_2&quot; type=&quot;long&quot; visibility=&quot;public&quot;/&gt; &lt;/valuetype&gt;</td>
</tr>
<tr>
<td>typedef short ShortType;</td>
<td>&lt;typedef name=&quot;ShortType&quot; type=&quot;short&quot;/&gt;</td>
</tr>
<tr>
<td>struct PrimitiveStruct { short short_member; } typedef PrimitiveStruct PrimitiveStructType;</td>
<td>&lt;struct name=&quot;PrimitiveStruct&quot;&gt; &lt;member name=&quot;short_member&quot; type=&quot;short&quot;/&gt; &lt;/struct&gt; &lt;typedef name=&quot;PrimitiveStructType&quot; nonBasic nonBasicTypeName=&quot;PrimitiveStruct&quot;/&gt;</td>
</tr>
</tbody>
</table>
### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XML</strong></td>
</tr>
<tr>
<td>IDL</td>
<td>XML</td>
</tr>
<tr>
<td>IDL</td>
<td>XML</td>
</tr>
</tbody>
</table>
| structs | <struct name="OneArrayStruct">
|  | <member name="short_array"
|  | type="short" arrayDimensions="2"/>
|  | </struct> |
| structs | <struct name="TwoArrayStruct">
|  | <member name="short_array"
|  | type="short" arrayDimensions="1,2"/>
|  | </struct> |
| structs | <struct name="SequenceStruct">
|  | <member name="short_sequence"
|  | type="short" sequenceMaxLength="4"/>
|  | </struct> |
| structs | <struct name="SequenceStruct">
|  | <member name="short_sequence"
|  | type="short" sequenceMaxLength="-1"/>
|  | </struct> |
| typedefs | <typedef name="ShortArray"
|  | type="short" dimensions="2"/>
|  | </typedef> |
| typedefs | <typedef name="SequenceOfArrayStruct">
|  | <member name="short_array_sequence"
|  | type="nonBasic"
|  | nonBasicTypeName="ShortSequence"
|  | sequenceMaxLength="2"/>
|  | </typedef> |
| typedefs | <typedef name="SequenceOfSequencesStruct">
|  | <member name="short_sequence_sequence"
|  | type="nonBasic"
|  | nonBasicTypeName="ShortSequence"
|  | sequenceMaxLength="2"/>
|  | </typedef> |

**Arrays**

<table>
<thead>
<tr>
<th>Attribute arrayDimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
</tr>
<tr>
<td>IDL</td>
</tr>
<tr>
<td>IDL</td>
</tr>
</tbody>
</table>

**Bounding Sequence**

<table>
<thead>
<tr>
<th>Attribute sequenceMaxLength &gt; 0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
</tr>
<tr>
<td>IDL</td>
</tr>
<tr>
<td>IDL</td>
</tr>
</tbody>
</table>

**Unbounding Sequence**

<table>
<thead>
<tr>
<th>Attribute sequenceMaxLength set to -1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
</tr>
<tr>
<td>IDL</td>
</tr>
<tr>
<td>IDL</td>
</tr>
</tbody>
</table>

**Array of Sequences**

<table>
<thead>
<tr>
<th>Attributes sequenceMaxLength and arrayDimensions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
</tr>
<tr>
<td>IDL</td>
</tr>
<tr>
<td>IDL</td>
</tr>
</tbody>
</table>

**Sequence of Arrays**

<table>
<thead>
<tr>
<th>Must be implemented with a typedef tag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
</tr>
<tr>
<td>IDL</td>
</tr>
<tr>
<td>IDL</td>
</tr>
</tbody>
</table>

**Sequence of Sequences**

<table>
<thead>
<tr>
<th>Must be implemented with a typedef tag</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
</tr>
<tr>
<td>IDL</td>
</tr>
<tr>
<td>IDL</td>
</tr>
</tbody>
</table>
### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XML</strong></td>
</tr>
<tr>
<td>module</td>
<td>module tag</td>
</tr>
<tr>
<td>struct PrimitiveStruct { long long_member; }</td>
<td>&lt;struct name=&quot;PrimitiveStruct&quot;&gt; &lt;member name=&quot;long_member&quot; type=&quot;long&quot;/&gt; &lt;/struct&gt;</td>
</tr>
<tr>
<td>include</td>
<td>include tag (works only within the &lt;types&gt; tag to include types from different XML files)</td>
</tr>
<tr>
<td>#include &quot;PrimitiveTypes.idl&quot;</td>
<td>&lt;include file=&quot;PrimitiveTypes.xml&quot;/&gt;</td>
</tr>
<tr>
<td>@key annotation 1</td>
<td>key attribute with values true, false, 0, or 1 Default (if not present): 0</td>
</tr>
<tr>
<td>struct KeyedPrimitiveStruct { @key short short_member; }</td>
<td>&lt;struct name=&quot;KeyedPrimitiveStruct&quot;&gt; &lt;member name=&quot;short_member&quot; type=&quot;short&quot; key=&quot;true&quot;/&gt; &lt;/struct&gt;</td>
</tr>
<tr>
<td>@external or pointer</td>
<td>external attribute with values true, false, 0, or 1 Default (if not present): 0</td>
</tr>
<tr>
<td>struct PrimitiveStruct { @external long long_member; }</td>
<td>&lt;struct name=&quot;PointerStruct&quot;&gt; &lt;member name=&quot;long_member&quot; type=&quot;long&quot; external=&quot;true&quot;/&gt; &lt;/struct&gt;</td>
</tr>
<tr>
<td>@optional annotation</td>
<td>optional attribute with values true, false, 0, or 1 Default (if not present): 0</td>
</tr>
<tr>
<td>struct Point { long x; long y; @optional long z; }</td>
<td>&lt;struct name=&quot;Point&quot;&gt; &lt;member name=&quot;x&quot; type=&quot;int32&quot;/&gt; &lt;member name=&quot;y&quot; type=&quot;int32&quot;/&gt; &lt;member name=&quot;z&quot; type=&quot;int32&quot; optional=&quot;true&quot;/&gt; &lt;/struct&gt;</td>
</tr>
</tbody>
</table>

1For information on this and the other annotations, see 3.3.9 Using Builtin Annotations on page 112.
### 3.4 Creating User Data Types with Extensible Markup Language (XML)

#### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
</table>
| **@id annotation** | @mutable struct Point {
| | id(56) long x;
| | id(57) long y;
| | long z;
| | };
| | <struct name="Point" extensibility="mutable">
| | <member name="x" id="56" type="long"/>
| | <member name="y" id="57" type="long"/>
| | <!-- z id is 58 -->
| | <member name="y" type="long"/>
| | </struct> |
| **@hashid annotation** | @mutable struct Point {
| | hashid long x;
| | hashid("other_y") long y;
| | };
| | <struct name="Point" extensibility="mutable">
| | <member name="x" hashid="x" type="int32"/>
| | <member name="y" hashid="other_y" type="int32"/>
| | </struct> |
| **@value annotation** | enum PrimitiveEnum {
| | @value (10) ENUM1,
| | @value (20) ENUM2,
| | ENUM3
| | };
| | <enum name="PrimitiveEnum">
| | <enumerator name="ENUM1" value="10"/>
| | <enumerator name="ENUM2" value="20"/>
| | <!-- ENUM3 id is 21 -->
| | <enumerator name="ENUM3"/>
| | </enum> |
| **@defaultLiteral annotation** | enum MyEnum {
| | ENUM1,
| | @defaultLiteral ENUM2
| | };
| | <enum name="MyEnum">
| | <enumerator name="ENUM1"/>
| | <enumerator name="ENUM2" defaultLiteral="true"/>
| | </enum> |
### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>@default annotation</td>
<td>@default(24) typedef long MyLongTypeDefWithDefault; struct Point { @default(42) long x; MyLongTypeDefWithDefault y; } &lt;typedef name=&quot;MyLongTypeDefWithDefault&quot; type=&quot;long&quot; default=&quot;24&quot;/&gt; &lt;struct name=&quot;Point&quot;&gt; &lt;member name=&quot;x&quot; type=&quot;long&quot; default=&quot;42&quot;/&gt; &lt;member name=&quot;y&quot; type=&quot;nonBasic&quot; nonBasicTypeName=&quot;MyLongTypeDefWithDefault&quot;/&gt; &lt;/struct&gt;</td>
</tr>
<tr>
<td>@min annotation</td>
<td>struct Point { @min(-32) long x; long y; } &lt;struct name=&quot;Point&quot;&gt; &lt;member name=&quot;x&quot; type=&quot;long&quot; min=&quot;-32&quot;/&gt; &lt;member name=&quot;y&quot; type=&quot;long&quot;/&gt; &lt;/struct&gt;</td>
</tr>
<tr>
<td>@max annotation</td>
<td>struct Point { @max(31) long x; long y; } &lt;struct name=&quot;Point&quot;&gt; &lt;member name=&quot;x&quot; type=&quot;long&quot; max=&quot;31&quot;/&gt; &lt;member name=&quot;y&quot; type=&quot;long&quot;/&gt; &lt;/struct&gt;</td>
</tr>
<tr>
<td>@range annotation</td>
<td>Not supported. Use min and max attributes instead. struct Point { @range(min = -32, max = 31) long x; long y; } &lt;struct name=&quot;Point&quot;&gt; &lt;member name=&quot;x&quot; type=&quot;long&quot; min=&quot;-32&quot; max=&quot;31&quot;/&gt; &lt;member name=&quot;y&quot; type=&quot;long&quot;/&gt; &lt;/struct&gt;</td>
</tr>
</tbody>
</table>
### 3.4 Creating User Data Types with Extensible Markup Language (XML)

#### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XML</strong></td>
</tr>
<tr>
<td>@autoid annotation</td>
<td>@mutable @hashid(HASH) struct Point { long x; long y; }</td>
</tr>
<tr>
<td></td>
<td>&lt;struct name=&quot;Point&quot; extensibility=&quot;mutable&quot; autoid=&quot;hash&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;x&quot; type=&quot;long&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;y&quot; type=&quot;long&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
</tr>
<tr>
<td>@nested annotation</td>
<td>@mutable @nest TopLevelPrimitiveStruct { short short_member; }</td>
</tr>
<tr>
<td></td>
<td>&lt;struct name=&quot;TopLevelPrimitiveStruct&quot; nested=&quot;true&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;short_member&quot; type=&quot;short&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
</tr>
<tr>
<td>@mutable, @appendable, or @final annotation</td>
<td>@mutable struct Point { long x; long y; }</td>
</tr>
<tr>
<td></td>
<td>&lt;struct name=&quot;Point&quot; extensibility=&quot;mutable&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;x&quot; type=&quot;long&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;y&quot; type=&quot;long&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
</tr>
</tbody>
</table>
### Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XML</td>
</tr>
<tr>
<td>allowed_data_representation attribute with values xcdr, xcdr2, or xml</td>
<td></td>
</tr>
<tr>
<td>Default (if not present): xcdr2 for FlatData language binding; the allowed_data_representation value in ancestor module(s) or (xcdr</td>
<td>xcdr2) for plain language binding</td>
</tr>
<tr>
<td>IDL</td>
<td>XML</td>
</tr>
</tbody>
</table>
| @allowed_data_representation(XCDR2) @mutable struct Point { long x; long y; } | <struct name="Point" extensibility= "mutable" allowed_data_representation="xcdr2">
|                                                     |         |
|                                                     |         |
| @use_vector attribute with values true, false, 0, or 1 | struct Image { @use_vector sequence<octet, 1048576> pixels; } | <struct name="Image">
| Default (if not present): false unless code generated with -alwaysUseStdVector |         |
| languageBinding attribute with values plain or flat data. Default (if not present): the languageBinding value in ancestor module(s) or plain if not specified |         |
| IDL                                                 | XML     |
| @language_binding (FLAT_DATA) @final struct Point { long x; long y; } | <struct name="Point" extensibility= "final" languageBinding="flat_data">
|                                                     |         |
Table 3.12 Mapping Type System Constructs to XML

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XML</td>
</tr>
<tr>
<td>transferMode attribute</td>
<td>@trans-</td>
</tr>
<tr>
<td>with values in-band or</td>
<td>fer_mode</td>
</tr>
<tr>
<td>shmem_ref.</td>
<td>(SHMEM_</td>
</tr>
<tr>
<td>Default (if not present):</td>
<td>REF)</td>
</tr>
<tr>
<td>the @transfer_mode</td>
<td>struct Point {</td>
</tr>
<tr>
<td>value in ancestor module(s) or inband if not specified</td>
<td>long x;</td>
</tr>
<tr>
<td></td>
<td>long y;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>&lt;struct name=&quot;Point&quot; transferMode=&quot;shmem_ref&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;x&quot; type=&quot;long&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;y&quot; type=&quot;long&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
</tr>
<tr>
<td>resolveName attribute</td>
<td>struct</td>
</tr>
<tr>
<td>with values true, false, 0, or 1</td>
<td>UnresolvedPrimitiveStruct {</td>
</tr>
<tr>
<td>Default (if not present):</td>
<td>@resolve_name(false)</td>
</tr>
<tr>
<td>@resolve_name of the parent type or false if not specified on parent</td>
<td>PrimitiveStruct</td>
</tr>
<tr>
<td></td>
<td>primitive_member;</td>
</tr>
<tr>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td>&lt;struct name=&quot;UnresolvedPrimitiveStruct&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;member name=&quot;primitive_member&quot; type=&quot;PrimitiveStruct&quot; resolveName=&quot;false&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;/struct&gt;</td>
</tr>
<tr>
<td>Other annotations</td>
<td>//@copy (This text will be copied in the generated files)</td>
</tr>
<tr>
<td>directive tag</td>
<td>&lt;directive kind=&quot;copy&quot;&gt;</td>
</tr>
<tr>
<td></td>
<td>This text will be copied in the generated files</td>
</tr>
<tr>
<td></td>
<td>&lt;/directive&gt;</td>
</tr>
</tbody>
</table>

3.5 Creating User Data Types with XML Schemas (XSD)

You can describe data types with XML schemas (XSD). The format is based on the standard IDL-to-WSDL mapping described in the OMG document "CORBA to WSDL/SOAP Interworking Specification."

Example Header for XSD:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:dds="http://www.omg.org/dds"
    xmlns:tns="http://www.omg.org/IDL-Mapped/"
    targetNamespace="http://www.omg.org/IDL-Mapped/"
    <xsd:import namespace="http://www.omg.org/dds"
        schemaLocation="rti.dds_topic_types_common.xsd"/>
```
3.5 Creating User Data Types with XML Schemas (XSD)

...<xsd:schema>

Table 3.13 Mapping Type System Constructs to XSD describes how to map IDL types to XSD. The Connext DDS code generator, riddsgen, will only accept XSD files that follow this mapping.

Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
</table>
| char           | dds:char<sup>1</sup> | struct PrimitiveStruct
char char_member; |
|               |     |     |     |     |
|               |     |     |     |     |
| wchar          | dds:wchar<sup>2</sup> | struct PrimitiveStruct
wchar wchar_member; |
|               |     |     |     |     |
|               |     |     |     |     |
| octet          | xsd:unsignedByte | struct PrimitiveStruct
octet octet_member; |
|               |     |     |     |     |
|               |     |     |     |     |
| int8<sup>3</sup> | dds:int8<sup>4</sup> | struct PrimitiveStruct
int8 int8_member; |
|               |     |     |     |     |

1All files that use the primitive types char, wchar, int8, uint8, long, double, and wstring must reference rti.dds_topic_types_common.xsd. See 3.5.1 Primitive Types on page 151.

2All files that use the primitive types char, wchar, int8, uint8, long, double, and wstring must reference rti.dds_topic_types_common.xsd. See 3.5.1 Primitive Types on page 151.

3This type is supported only at the API level. It is still considered an octet for type matching purposes.

4All files that use the primitive types char, wchar, int8, uint8, long, double, and wstring must reference rti.dds_topic_types_common.xsd. See 3.5.1 Primitive Types on page 151.
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8(^1)</td>
<td>dds:uint8(^2)</td>
<td>struct PrimitiveStruct ( uint8 uint8_member; )</td>
<td></td>
<td>&lt;xsd:complexType name=&quot;PrimitiveStruct&quot;&gt; <a href="">xsd:sequence</a> &lt;xsd:element name=&quot;uint8_member&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;dds:uint8&quot;/&gt; &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;</td>
</tr>
<tr>
<td>int16 or short</td>
<td>xsd:short</td>
<td>struct PrimitiveStruct ( int16 short_member; )</td>
<td></td>
<td>&lt;xsd:complexType name=&quot;PrimitiveStruct&quot;&gt; <a href="">xsd:sequence</a> &lt;xsd:element name=&quot;short_member&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:short&quot;/&gt; &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;</td>
</tr>
<tr>
<td>uint16 or unsigned short</td>
<td>xsd:unsignedShort</td>
<td>struct PrimitiveStruct ( uint16 unsigned_short_member; )</td>
<td></td>
<td>&lt;xsd:complexType name=&quot;PrimitiveStruct&quot;&gt; <a href="">xsd:sequence</a> &lt;xsd:element name=&quot;unsigned_short_member&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:unsignedShort&quot;/&gt; &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;</td>
</tr>
<tr>
<td>int32 or long</td>
<td>xsd:int</td>
<td>struct PrimitiveStruct ( int32 long_member; )</td>
<td></td>
<td>&lt;xsd:complexType name=&quot;PrimitiveStruct&quot;&gt; <a href="">xsd:sequence</a> &lt;xsd:element name=&quot;long_member&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt; &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;</td>
</tr>
<tr>
<td>uint32 or unsigned long</td>
<td>xsd:unsignedInt</td>
<td>struct PrimitiveStruct ( uint32 unsigned_long_member; )</td>
<td></td>
<td>&lt;xsd:complexType name=&quot;PrimitiveStruct&quot;&gt; <a href="">xsd:sequence</a> &lt;xsd:element name=&quot;unsigned_long_member&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:unsignedInt&quot;/&gt; &lt;/xsd:sequence&gt; &lt;/xsd:complexType&gt;</td>
</tr>
</tbody>
</table>

---

1. This type is supported only at the API level. It is still considered an octet for type matching purposes.

2. All files that use the primitive types char, wchar, int8, uint8, long double, and wstring must reference rti.dds_topic_types_common.xsd. See 3.5.1 Primitive Types on page 151.
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
</table>
| int64 or long long | struct PrimitiveStruct {
|                | int64 long_long_member;
|                | } | <xsd:complexType name="PrimitiveStruct">
|                | <xsd:sequence> | <xsd:element name="long_long_member">
|                | minOccurs="1" maxOccurs="1" | type="xsd:long"/>
|                | </xsd:sequence> | </xsd:complexType> |
| uint64 or unsigned long long | struct PrimitiveStruct {
|                | uint64 unsigned_long_long_member;
|                | } | <xsd:complexType name="PrimitiveStruct">
|                | <xsd:sequence> | <xsd:element name="unsigned_long_long_member">
|                | minOccurs="1" maxOccurs="1" | type="xsd:unsignedLong"/>
|                | </xsd:sequence> | </xsd:complexType> |
| float | struct PrimitiveStruct {
|                | float float_member;
|                | } | <xsd:complexType name="PrimitiveStruct">
|                | <xsd:sequence> | <xsd:element name="float_member">
|                | minOccurs="1" maxOccurs="1" | type="xsd:float"/>
|                | </xsd:sequence> | </xsd:complexType> |
| double | struct PrimitiveStruct {
|                | double double_member;
|                | } | <xsd:complexType name="PrimitiveStruct">
|                | <xsd:sequence> | <xsd:element name="double_member">
|                | minOccurs="1" maxOccurs="1" | type="xsd:double"/>
|                | </xsd:sequence> | </xsd:complexType> |
| long double | struct PrimitiveStruct {
|                | long double long_double_member;
|                | } | <xsd:complexType name="PrimitiveStruct">
|                | <xsd:sequence> | <xsd:element name="long_double_member">
|                | minOccurs="1" maxOccurs="1" | type="dds:longDouble"/>
|                | </xsd:sequence> | </xsd:complexType> |
| boolean | struct PrimitiveStruct {
|                | boolean boolean_member;
|                | } | <xsd:complexType name="PrimitiveStruct">
|                | <xsd:sequence> | <xsd:element name="boolean_member">
|                | minOccurs="1" maxOccurs="1" | type="xsd:boolean"/>
|                | </xsd:sequence> | </xsd:complexType> |
| unbounded string | struct PrimitiveStruct {
|                | string string_member;
|                | } | <xsd:complexType name="PrimitiveStruct">
|                | <xsd:sequence> | <xsd:element name="string_member">
|                | minOccurs="1" maxOccurs="1" | type="xsd:string"/>
|                | </xsd:sequence> | </xsd:complexType> |
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XSD</strong></td>
</tr>
<tr>
<td><code>xsd:string</code> with restriction to specify maximum length</td>
<td><code>&lt;xsd:complexType name=&quot;PrimitiveStruct_string_member_BoundedString&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>struct PrimitiveStruct { string&lt;20&gt; string_member; }</code></td>
<td><code>&lt;xsd:complexType name=&quot;PrimitiveStruct&quot;&gt;</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Related Information

1. All files that use the primitive types char, wchar, int8, uint8, long double and wstring must reference rti.dds.topic.types.common.xsd. See 3.5.1 Primitive Types on page 151.
## Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td></td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
<tr>
<td><strong>IDL</strong></td>
<td></td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
</tbody>
</table>

### Examples

**IDL**

```idl
struct
  PrimitiveStruct {
    wstring<20> wstring_member;
  }
```

**XSD**

```xsd
<xs:complexType name="PrimitiveStruct_wstring_member_BoundedString">
  <xs:element name="item" minOccurs="1" maxOccurs="1">
    <xs:simpleType>
      <xs:restriction base="dds:wstring">
        <xs:maxLength value="20" fixed="true"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:element>
</xs:complexType>
```

**IDL**

```idl
enum PrimitiveEnum {
  ENUM1,
  ENUM2,
  ENUM3
}
```

**XSD**

```xsd
<xs:simpleType name="PrimitiveEnum">
  <xs:restriction base="xsd:string">
    <xs:enumeration value="ENUM1"/>
    <xs:enumeration value="ENUM2"/>
    <xs:enumeration value="ENUM3"/>
  </xs:restriction>
</xs:simpleType>
```
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td></td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
</tbody>
</table>
| IDL            | enum PrimitiveEnum {  
|                | ENUM1 = 10,  
|                | ENUM2 = 20,  
|                | ENUM3 = 30 } |
|                | <xsd:simpleType name="PrimitiveEnum">  
|                | <xsd:restriction base="xsd:string">  
|                | <xsd:enumeration value="ENUM1">  
|                | <xsd:annotation>  
|                | <xsd:appinfo>  
|                | <ordinal>10</ordinal>  
|                | </xsd:appinfo>  
|                | </xsd:annotation>  
|                | </xsd:enumeration>  
|                | <xsd:enumeration value="ENUM2">  
|                | <xsd:annotation>  
|                | <xsd:appinfo>  
|                | <ordinal>20</ordinal>  
|                | </xsd:appinfo>  
|                | </xsd:annotation>  
|                | </xsd:enumeration>  
|                | <xsd:enumeration value="ENUM3">  
|                | <xsd:annotation>  
|                | <xsd:appinfo>  
|                | <ordinal>30</ordinal>  
|                | </xsd:appinfo>  
|                | </xsd:annotation>  
|                | </xsd:enumeration>  
|                | </xsd:restriction>  
|                | </xsd:simpleType> |
| IDL            | struct PrimitiveStruct {  
|                | short short_member;  
|                | } |
|                | <xsd:complexType name="PrimitiveStruct">  
|                | <xsd:sequence>  
|                | <xsd:element name="short_member"  
|                | minOccurs="1" maxOccurs="1"  
|                | type="xsd:short"/>  
|                | </xsd:element>  
|                | </xsd:sequence>  
|                | </xsd:complexType> |

**constant**  
IDL constants are mapped by substituting their value directly in the generated file.

**struct**  
xsd:-complexType with xsd:sequence
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>union</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xsd:-complexType with xsd:choice</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>union PrimitiveUnion switch (long) { case 1: short short_member; default: long long_member; }</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```xml
<xs:complexType name="PrimitiveUnion">
  <xs:sequence>
    <xs:element name="discriminator" type="xs:int"/>
    <xs:choice>
      <!-- case 1 -->
      <xs:element name="short_member" minOccurs="0" maxOccurs="1" type="xs:short">
        <xs:annotation>
          <xs:appinfo>
            <case>1</case>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
      <!-- case default -->
      <xs:element name="long_member" minOccurs="0" maxOccurs="1" type="xs:int">
        <xs:annotation>
          <xs:appinfo>
            <case>default</case>
          </xs:appinfo>
        </xs:annotation>
      </xs:element>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

1 The discriminant values can be described using comments (as specified by the standard) or xsd:annotation tags. We recommend using annotations because comments may be removed by XSD/XML parsers.
## Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XSD</strong></td>
</tr>
<tr>
<td><strong>typedef</strong></td>
<td></td>
</tr>
<tr>
<td>ShortType;</td>
<td></td>
</tr>
<tr>
<td>PrimitiveStruct (</td>
<td></td>
</tr>
<tr>
<td>short short_member;</td>
<td></td>
</tr>
<tr>
<td>)</td>
<td></td>
</tr>
<tr>
<td>typedef PrimitiveType =</td>
<td></td>
</tr>
<tr>
<td>PrimitiveStructType;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>typedef</strong></td>
<td></td>
</tr>
<tr>
<td>DerivedValueType:</td>
<td></td>
</tr>
<tr>
<td>BaseValueType {</td>
<td></td>
</tr>
<tr>
<td>public long long_</td>
<td></td>
</tr>
<tr>
<td>member;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>valuetype</td>
<td></td>
</tr>
<tr>
<td>DerivedValueType:</td>
<td></td>
</tr>
<tr>
<td>BaseValueType {</td>
<td></td>
</tr>
<tr>
<td>public long long_</td>
<td></td>
</tr>
<tr>
<td>member2;</td>
<td></td>
</tr>
<tr>
<td>public long long_</td>
<td></td>
</tr>
<tr>
<td>member3;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>valuetype</td>
<td></td>
</tr>
<tr>
<td>BaseType {</td>
<td></td>
</tr>
<tr>
<td>public long long_</td>
<td></td>
</tr>
<tr>
<td>member;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>valuetype</td>
<td></td>
</tr>
<tr>
<td>DerivedValueType:</td>
<td></td>
</tr>
<tr>
<td>BaseType {</td>
<td></td>
</tr>
<tr>
<td>public long long_</td>
<td></td>
</tr>
<tr>
<td>member2;</td>
<td></td>
</tr>
<tr>
<td>public long long_</td>
<td></td>
</tr>
<tr>
<td>member3;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>xsd:complexType</td>
<td></td>
</tr>
<tr>
<td>name=&quot;BaseValueType&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:element name=&quot;long_member&quot;</td>
<td></td>
</tr>
<tr>
<td>maxOccurs=&quot;1&quot; minOccurs=&quot;1&quot;</td>
<td></td>
</tr>
<tr>
<td>type=&quot;xs:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!- @visibility public --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:complexType&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>xsd:complexType</td>
<td></td>
</tr>
<tr>
<td>name=&quot;DerivedValueType&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:extension base=&quot;BaseValueType&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:element name=&quot;long_member2&quot;</td>
<td></td>
</tr>
<tr>
<td>maxOccurs=&quot;1&quot; minOccurs=&quot;1&quot;</td>
<td></td>
</tr>
<tr>
<td>type=&quot;xs:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!- @visibility public --&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:element name=&quot;long_member3&quot;</td>
<td></td>
</tr>
<tr>
<td>maxOccurs=&quot;1&quot; minOccurs=&quot;1&quot;</td>
<td></td>
</tr>
<tr>
<td>type=&quot;xs:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!- @visibility public --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:extension&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:complexType&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!- @valuetype true --&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:complexType</td>
<td></td>
</tr>
<tr>
<td>name=&quot;DerivedValueType&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:extension base=&quot;BaseValueType&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:element name=&quot;long_member2&quot;</td>
<td></td>
</tr>
<tr>
<td>maxOccurs=&quot;1&quot; minOccurs=&quot;1&quot;</td>
<td></td>
</tr>
<tr>
<td>type=&quot;xs:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!- @visibility public --&gt;</td>
<td></td>
</tr>
<tr>
<td>xsd:element name=&quot;long_member3&quot;</td>
<td></td>
</tr>
<tr>
<td>maxOccurs=&quot;1&quot; minOccurs=&quot;1&quot;</td>
<td></td>
</tr>
<tr>
<td>type=&quot;xs:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!- @visibility public --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:extension&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:complexType&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!- @valuetype true --&gt;</td>
<td></td>
</tr>
<tr>
<td>typedef short short</td>
<td></td>
</tr>
<tr>
<td>PrimitiveType =</td>
<td></td>
</tr>
<tr>
<td>PrimitiveStructType;</td>
<td></td>
</tr>
<tr>
<td>typedef short short</td>
<td></td>
</tr>
<tr>
<td>PrimitiveType =</td>
<td></td>
</tr>
<tr>
<td>PrimitiveStructType;</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XSD</strong></td>
</tr>
<tr>
<td><strong>arrays</strong></td>
<td></td>
</tr>
<tr>
<td>n xsd:-complexType with sequence containing one element with min &amp; max occurs</td>
<td>There is one xsd:-complexType per array dimension</td>
</tr>
</tbody>
</table>
| struct OneArrayStruct { short short_array [2]; } |  \[
| <!-- Array type --> |
| <xsd:complexType name="OneArrayStruct_short_array_ArrayOfShort"> |
| <xsd:sequence> |
| <xsd:element name="item" minOccurs="2" maxOccurs="2" type="xsd:short"> |
| </xsd:element> |
| </xsd:sequence> |
| </xsd:complexType> |
| <!-- Struct w unidimensional array member --> |
| <xsd:complexType name="OneArrayStruct"> |
| <xsd:sequence> |
| <xsd:element name="short_array" minOccurs="1" maxOccurs="1" type="OneArrayStruct_short_array_ArrayOfShort"/> |
| </xsd:sequence> |
| </xsd:complexType> |
| **arrays (cont'd)** | | | |
| n xsd:-complexType with sequence containing one element with min & max occurs | There is one xsd:-complexType per array dimension |
| struct TwoArrayStruct { short short_array [2][1]; } |  \[
| <!--Second dimension array type --> |
| <xsd:complexType name="TwoArrayStruct_short_array_ArrayOfArrayOfShort"> |
| <xsd:sequence> |
| <xsd:element name="item" minOccurs="2" maxOccurs="2" type="xsd:short"> |
| </xsd:element> |
| </xsd:sequence> |
| </xsd:complexType> |
| <!-- First dimension array type --> |
| <xsd:complexType name="TwoArrayStruct_short_array_ArrayOfArrayOfArrayOfShort"> |
| <xsd:sequence> |
| <xsd:element name="item" minOccurs="1" maxOccurs="1" type="TwoArrayStruct_short_array_ArrayOfArrayOfArrayOfShort"/> |
| </xsd:element> |
| </xsd:sequence> |
| </xsd:complexType> |
| <!--Struct containing a bidimensional array member --> |
| <xsd:complexType name="TwoArrayStruct"> |
| <xsd:sequence> |
| <xsd:element name="short_array" minOccurs="1" maxOccurs="1" type="TwoArrayStruct_short_array_ArrayOfArrayOfArrayOfShort"/> |
| </xsd:element> |
| </xsd:sequence> |
| </xsd:complexType> |
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XSD</td>
</tr>
<tr>
<td><strong>Bound-ed sequence</strong></td>
<td></td>
</tr>
<tr>
<td>IDL</td>
<td></td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
<tr>
<td>IDL</td>
<td></td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Un-bound-ed sequence</strong></td>
<td></td>
</tr>
<tr>
<td>IDL</td>
<td></td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
<tr>
<td>IDL</td>
<td></td>
</tr>
<tr>
<td>XSD</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```xml
<!-- Sequence type -->
<xsd:complexType name="SequenceStruct_short_sequence_SequenceOfShort">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="4" type="xsd:short" />
  </xsd:sequence>
</xsd:complexType>

<!-- Struct containing a bounded sequence member -->
<xsd:complexType name="SequenceStruct">
  <xsd:sequence>
    <xsd:element name="short_sequence" minOccurs="1" maxOccurs="1" type= "SequenceStruct_short_sequence_SequenceOfShort"/>
  </xsd:sequence>
</xsd:complexType>

<!-- Sequence type -->
<xsd:complexType name="SequenceStruct_short_sequence_SequenceOfShort">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="unbounded" type="xsd:short" />
  </xsd:sequence>
</xsd:complexType>

<!-- Struct containing unbounded sequence member -->
<xsd:complexType name="SequenceStruct">
  <xsd:sequence>
    <xsd:element name="short_sequence" minOccurs="1" maxOccurs="1" type= "SequenceStruct_short_sequence_SequenceOfShort"/>
  </xsd:sequence>
</xsd:complexType>
```
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>Example</td>
</tr>
<tr>
<td>XSD</td>
<td>Example</td>
</tr>
<tr>
<td>IDL</td>
<td>Example</td>
</tr>
<tr>
<td>XSD</td>
<td>Example</td>
</tr>
<tr>
<td>n + 1 xsd-complexType with sequence containing one element with min &amp; max occurrences. There is one xsd-complexType per array dimension and one xsd-complexType for the sequence.</td>
<td>Example</td>
</tr>
<tr>
<td>array of sequences</td>
<td>Example</td>
</tr>
</tbody>
</table>

```xml
<!-- Sequence declaration -->
<xsd:complexType name="ArrayOfSequencesStruct_sequence_array_SequenceOfShort">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="0" maxOccurs="4" type="xsd:short"/>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>

<!-- Array declaration -->
<xsd:complexType name="ArrayOfSequencesStruct_sequence_arrayArrayOfSequenceOfShort">
  <xsd:sequence>
    <xsd:element name="item" minOccurs="2" maxOccurs="2" type="ArrayOfSequencesStruct_sequence_arrayArrayOfSequenceOfShort"/>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>

<!-- Structure containing a member that is an array of sequences -->
<xsd:complexType name="ArrayOfSequencesStruct">
  <xsd:sequence>
    <xsd:element name="sequence_array" minOccurs="1" maxOccurs="1" type="ArrayOfSequencesStruct_sequence_arrayArrayOfSequenceOfShort"/>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
```
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XSD</td>
</tr>
<tr>
<td>Sequence of arrays</td>
<td>Sequences of arrays must be implemented using an explicit type definition (typedef) for the array</td>
</tr>
<tr>
<td>IDL</td>
<td>XSD</td>
</tr>
</tbody>
</table>
|                         | <!-- Array declaration -->
|                         | <xsd:complexType name="ShortArray">
|                         |  <xsd:element name="item" minOccurs="2" maxOccurs="2" type="xsd:short">                                                       |
|                         |  </xsd:element>                                                                                                                      |
|                         |  </xsd:sequence>                                                                                                                     |
|                         | </xsd:complexType>                                                                                                                   |
|                         | <!-- Sequence declaration -->
|                         | <xsd:complexType name="SequencesOfArraysStruct_array_sequence_SequenceOfShortArray">
|                         |  <xsd:element name="item" minOccurs="0" maxOccurs="2" type="ShortArray">                                                        |
|                         |  </xsd:element>                                                                                                                      |
|                         | </xsd:sequence>                                                                                                                      |
|                         | </xsd:complexType>                                                                                                                   |
|                         | <!-- Struct containing a sequence of arrays -->
|                         | <xsd:complexType name="SequenceOfArraysStruct">
|                         |  <xsd:element name="arrays_sequence" minOccurs="1" maxOccurs="1" type=""SequencesOfArraysStruct_array_sequence_SequenceOfShortArray"> |
|                         |  </xsd:element>                                                                                                                      |
|                         | </xsd:sequence>                                                                                                                      |
|                         | </xsd:complexType>                                                                                                                   |
### 3.5 Creating User Data Types with XML Schemas (XSD)

#### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XSD</strong></td>
</tr>
</tbody>
</table>
| `typedef` sequence<short,4> ShortSequence; | ```xml
  <!-- Internal sequence declaration -->
  <xsd:complexType name="ShortSequence">
    <xsd:sequence>
      <xsd:element name="item"
        minOccurs="0" maxOccurs="4"
        type="xsd:short"/>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>
```
| Nothing |
| Module Modules are mapped adding the name of the module before the name of each type inside the module | ```xml
  <xsd:complexType name="PackageName.PrimitiveStruct">
    <xsd:sequence>
      <xsd:element name="long_member"
        minOccurs="1" maxOccurs="1"
        type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```
| Nothing |
| `include` | ```xml
  #include "PrimitiveType.idl"
```
| Nothing |
| **@key annotation** | ```xml
  struct KeyedPrimitiveStruct {
    @key short short_member;
  };
```
| ```xml
  <xsd:complexType name="KeyedPrimitiveStruct">
    <xsd:sequence>
      <xsd:element name="long_member"
        minOccurs="1" maxOccurs="1"
        type="xsd:int"/>
    </xsd:sequence>
</xsd:complexType>
```

*For information on this and the other annotations, see [3.3.9 Using Builtin Annotations on page 112](#)*
### 3.5 Creating User Data Types with XML Schemas (XSD)

Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XSD</strong></td>
</tr>
<tr>
<td>&lt;!-- @external &lt;true</td>
<td>-false]&gt;1[0]&gt; ---&gt;</td>
</tr>
<tr>
<td>Default if not specified: false</td>
<td></td>
</tr>
<tr>
<td>struct PrimitiveStruct {</td>
<td></td>
</tr>
<tr>
<td>long long_member;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:complexType name=&quot;PrimitiveStruct&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;long_member&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @external true --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:complexType&gt;</td>
<td></td>
</tr>
<tr>
<td><strong>minOccurs attribute set to 0 or 1</strong></td>
<td></td>
</tr>
<tr>
<td>Default (if not present): 1</td>
<td></td>
</tr>
<tr>
<td>struct Point {</td>
<td></td>
</tr>
<tr>
<td>long x;</td>
<td></td>
</tr>
<tr>
<td>long y;</td>
<td></td>
</tr>
<tr>
<td>@optional long z;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:complexType name=&quot;Point&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;x&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @id 56 --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;y&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @id 57 --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;z&quot; minOccurs=&quot;0&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:complexType&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @mutable --&gt;</td>
<td></td>
</tr>
<tr>
<td>struct Point {</td>
<td></td>
</tr>
<tr>
<td>@id(56) long x;</td>
<td></td>
</tr>
<tr>
<td>@id(57) long y;</td>
<td></td>
</tr>
<tr>
<td>long z;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:complexType name=&quot;Point&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;x&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @id 56 --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;y&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @id 57 --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;z&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:complexType&gt;</td>
<td></td>
</tr>
<tr>
<td><strong>@mutable</strong></td>
<td></td>
</tr>
<tr>
<td>struct Point {</td>
<td></td>
</tr>
<tr>
<td>@hashid(long x);</td>
<td></td>
</tr>
<tr>
<td>@hashid(&quot;other_y&quot;) long y;</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:complexType name=&quot;Point&quot;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;x&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @hashid --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;y&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @hashid other_y --&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;xsd:element name=&quot;z&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:sequence&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;/xsd:complexType&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;!-- @extensibility mutable---&gt;</td>
<td></td>
</tr>
</tbody>
</table>
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IDL</strong></td>
<td><strong>XSD</strong></td>
</tr>
<tr>
<td>&lt;!-- @ordinal &lt;value&gt;--&gt;</td>
<td></td>
</tr>
<tr>
<td>Default (if not present): the value of the previous enumerator plus 1</td>
<td></td>
</tr>
<tr>
<td>enum PrimitiveEnum { @value (10) ENUM1, @value (20) ENUM2, ENUM3 }</td>
<td>&lt;xsd:simpleType name=&quot;PrimitiveEnum&quot;&gt; &lt;xsd:restriction base=&quot;xsd:string&quot;&gt; &lt;xsd:enumeration value=&quot;ENUM1&quot;&gt; &lt;/xsd:enumeration&gt; &lt;xsd:enumeration value=&quot;ENUM2&quot;&gt; &lt;/xsd:enumeration&gt; &lt;xsd:enumeration value=&quot;ENUM3&quot;&gt; &lt;/xsd:enumeration&gt; &lt;/xsd:restriction&gt; &lt;/xsd:simpleType&gt;</td>
</tr>
<tr>
<td>&lt;!-- @ordinal &lt;value&gt;--&gt;</td>
<td></td>
</tr>
<tr>
<td>Default (if not present): 0*</td>
<td></td>
</tr>
<tr>
<td>enum MyEnum { ENUM1, @default_literal ENUM2 }</td>
<td>&lt;xsd:simpleType name=&quot;MyEnum&quot;&gt; &lt;xsd:restriction base=&quot;xsd:string&quot;&gt; &lt;xsd:enumeration value=&quot;ENUM1&quot;&gt; &lt;/xsd:enumeration&gt; &lt;xsd:enumeration value=&quot;ENUM2&quot;&gt; &lt;/xsd:enumeration&gt; &lt;/xsd:restriction&gt; &lt;/xsd:simpleType&gt;</td>
</tr>
<tr>
<td>@default literal</td>
<td></td>
</tr>
<tr>
<td>default literal annotation with values true, false, 0, or 1</td>
<td></td>
</tr>
<tr>
<td>Default (if not present): 0*</td>
<td></td>
</tr>
<tr>
<td>default attribute for elements inside a structure, default literal annotation for type definitions</td>
<td></td>
</tr>
<tr>
<td>Default (if not present in this member or its alias types): 0, the empty string, or whichever enumerator is the default literal</td>
<td></td>
</tr>
</tbody>
</table>

#### XML Schemas (XSD)

```xml
<xs:simpleType name="PrimitiveEnum">
  <xs:restriction base="xsd:string">
    <xs:enumeration value="ENUM1"/>
    <xs:enumeration value="ENUM2"/>
    <xs:enumeration value="ENUM3"/>
  </xs:restriction>
</xs:simpleType>
```

```xml
<xs:simpleType name="MyEnum">
  <xs:restriction base="xsd:string">
    <xs:enumeration value="ENUM1"/>
  </xs:restriction>
</xs:simpleType>
```
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
</table>
| default attribute for alias elements inside a structure, default appinfo annotation for type definitions and regular strings inside a structure | @default("myDefault") typedef string MyStringTypeDefWithDefault; struct DefaultString {
| Default (if not present in this member or its alias types): 0, the empty string, or whichever enumerator is the default literal | @default("myDefault")
| string x;
| MyStringTypeDefWithDefault y;
| @default ("myDefaultDefault")
| MyStringTypeDefWithDefault z; | <xsd:simpleType name="MyStringTypeDefWithDefault">
| <xsd:restriction base="xsd:string">
| <xsd:annotation>
| <xsd:appinfo>
| <default>"myDefault"</default>
| </xsd:appinfo>
| </xsd:annotation>
| <xsd:maxLength value="255" fixed="true"/>
| </xsd:restriction>
| </xsd:simpleType>
| <xsd:complexType name="DefaultString_x_BoundedString">
| <xsd:sequence>
| <xsd:element name="item" minOccurs="1" maxOccurs="1">
| <xsd:simpleType>
| <xsd:restriction base="xsd:string">
| <xsd:maxLength value="255" fixed="true"/>
| </xsd:restriction>
| </xsd:simpleType>
| </xsd:element>
| </xsd:sequence>
| </xsd:complexType>
| <xsd:complexType name="DefaultString">
| <xsd:sequence>
| <xsd:element name="x" minOccurs="1" maxOccurs="1" type="tns:DefaultString_x_BoundedString"/>
| <xsd:annotation>
| <xsd:appinfo>
| <default>"string"</default>
| </xsd:appinfo>
| </xsd:annotation>
| </xsd:element>
| <xsd:element name="y" minOccurs="1" maxOccurs="1" type="tns:MyStringTypeDefWithDefault"/>
| <xsd:element name="z" minOccurs="1" maxOccurs="1" type="tns:MyStringTypeDefWithDefault" default="myDefaultDefault"/>
| </xsd:sequence>
| </xsd:complexType>
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XSD</td>
</tr>
</tbody>
</table>
| @min annotation | `<xsd:simpleType name="myLongDefault">`<br>`<xsd:restriction base="xsd:int">`<br>`<xsd:annotation>`<br>`<xsd:appinfo>`<br>`<min>-32</min>`<br>`</xsd:appinfo>`<br>`</xsd:restriction>`<br>`</xsd:simpleType>`<br>`<xsd:complexType name="Point">`<br>`<xsd:sequence>`<br>`<xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/>`<br>`<xsd:element name="y" minOccurs="1" maxOccurs="1" type="xsd:int"/>`<br>`<xsd:element name="myX" minOccurs="1" maxOccurs="1" type="tns:myLongDefault"/>`<br>`</xsd:sequence>`<br>`</xsd:complexType>`<br>`<min(-32)>`typedef long myLongDefault;`<br>`struct Point {`<br>`  @min(-32) long x;`<br>`  long y;`<br>`  myLongDefault myX;`<br>`};`<br>`@min(-32)`<br>`typedef long myLongDefault;`<br>`struct Point {`<br>`  @min(-32) long x;`<br>`  long y;`<br>`  myLongDefault myX;`<br>`};`<br>`©min annotation | `@max annotation` | `<xsd:simpleType name="myLongDefault">`<br>`<xsd:restriction base="xsd:int">`<br>`<xsd:annotation>`<br>`<xsd:appinfo>`<br>`<max>31</max>`<br>`</xsd:appinfo>`<br>`</xsd:restriction>`<br>`</xsd:simpleType>`<br>`<xsd:complexType name="Point">`<br>`<xsd:sequence>`<br>`<xsd:element name="x" minOccurs="1" maxOccurs="1" type="xsd:int"/>`<br>`<xsd:element name="y" minOccurs="1" maxOccurs="1" type="xsd:int"/>`<br>`<xsd:element name="myX" minOccurs="1" maxOccurs="1" type="tns:myLongDefault"/>`<br>`</xsd:sequence>`<br>`</xsd:complexType>`<br>`<max(31)>`typedef long myLongDefault;`<br>`struct Point {`<br>`  @max(31) long x;`<br>`  long y;`<br>`  myLongDefault myX;`<br>`};`<br>`@max annotation | **Table 3.13 Mapping Type System Constructs to XSD**

**3.5 Creating User Data Types with XML Schemas (XSD)**
Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XSD</td>
</tr>
<tr>
<td><code>@range(min = -32, max = 31)</code> typedef long myLongDefault; struct Point { @range(min = -32, max = 31) long x; long y; myLongDefault myX; }</td>
<td><code>&lt;xsd:simpleType name=&quot;myLongDefault&quot;&gt; &lt;xsd:restriction base=&quot;xsd:int&quot;&gt; &lt;xsd:appinfo&gt; &lt;min&gt;-32&lt;/min&gt; &lt;max&gt;31&lt;/max&gt; &lt;/xsd:appinfo&gt; &lt;/xsd:restriction&gt; &lt;/xsd:simpleType&gt;</code></td>
</tr>
<tr>
<td>`&lt;!- @autoid [hash</td>
<td>sequential] -&gt; Default if not present; the @autoid value in ancestor module(s) or sequential if not specified.`</td>
</tr>
<tr>
<td><code>&lt;!- @nested or @top-level annotation !-&gt; Default (if not specified): true</code></td>
<td><code>&lt;xsd:complexType name=&quot;TopLevelPrimitiveStruct&quot;&gt; { short short_member; }</code></td>
</tr>
</tbody>
</table>

```xml
<!- @autoid (HASH) -> @nested or @top-level annotation
<!- @topLevel [true|false]|0> !-> Default (if not specified): true
```
### Table 3.13 Mapping Type System Constructs to XSD

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>@extensibility, @mutable, @appendable, or @final annotation</td>
<td>&lt;!-- @extensibility &lt;final</td>
<td>appendable</td>
<td>mutable&gt; --&gt; Default (if not present); appendable</td>
<td>@mutable struct Point { long x; long y; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;xsd:element name=&quot;x&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td>&lt;xsd:element name=&quot;x&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;xsd:element name=&quot;y&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td>&lt;xsd:element name=&quot;y&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;/xsd:sequence&gt;</td>
<td>&lt;/xsd:sequence&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;/xsd:complexType&gt;</td>
<td>&lt;/xsd:complexType&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;!-- @struct true --&gt;</td>
<td>&lt;!-- @extensibility mutable --&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>@allowed_data_representation</td>
<td>&lt;!-- @allowed_data_representation &lt;xcdr</td>
<td>xcdr2&gt;</td>
<td>xml&gt; --&gt; Default (if not present); xcdr2 for flat data language binding; the @allowed_data_representation value in ancestor module(s) or (xcdr</td>
<td>xcdr2) for plain language</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;xsd:element name=&quot;x&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td>&lt;xsd:element name=&quot;x&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;xsd:element name=&quot;y&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
<td>&lt;xsd:element name=&quot;y&quot; minOccurs=&quot;1&quot; maxOccurs=&quot;1&quot; type=&quot;xsd:int&quot;/&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;/xsd:sequence&gt;</td>
<td>&lt;/xsd:sequence&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;/xsd:complexType&gt;</td>
<td>&lt;/xsd:complexType&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&lt;!-- @allowed_data_representation xcdr2 --&gt;</td>
<td>&lt;!-- @extensibility mutable --&gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>IDL</th>
<th>XSD</th>
<th>IDL</th>
<th>XSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>@use_vector annotation</td>
<td>&lt;!-- @use_vector &lt;true&gt;</td>
<td>false&gt;</td>
<td>1</td>
<td>0&gt; --&gt; Default (if not present); false unless code generated with -alwaysUseStdVector</td>
</tr>
</tbody>
</table>
### 3.5.1 Primitive Types

The primitive types char, wchar, long double, and wstring are not supported natively in XSD. Connext DDS provides definitions for these types in the file `<NDDSHOME>/resource/app/app_

---

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XSD</td>
</tr>
</tbody>
</table>
| <!-- @languageBinding <plain|flat-data> --> | <xsd:complexType name="Point">
| @language_binding flat-data |   @language_binding (FLAT_DATA)
| Default (if not present): the @language_binding value in ancestor module(s) or plain if not specified |   @final
| struct Point { |   long x;
|   long y; } |   <xsd:element name="x" minOccurs="1"
|                      |   maxOccurs="1" type="xsd:int"/>
|                      |   <xsd:element name="y" minOccurs="1"
|                      |   maxOccurs="1" type="xsd:int"/>
|                      | </xsd:sequence>
|                      | </xsd:complexType> |
| <!-- @struct true --> | <!-- @language_binding flat-data--> |
| <!-- @transfer_mode shmem_ref --> | <!-- @extensibility final--> |

---

**Table 3.13 Mapping Type System Constructs to XSD**

<table>
<thead>
<tr>
<th>Type/Construct</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDL</td>
<td>XSD</td>
</tr>
</tbody>
</table>
| <!-- @languageBinding <plain|flat-data> --> | <xsd:complexType name="Point">
| @language_binding flat-data |   @language_binding (FLAT_DATA)
| Default (if not present): the @language_binding value in ancestor module(s) or plain if not specified |   @final
| struct Point { |   long x;
|   long y; } |   <xsd:element name="x" minOccurs="1"
|                      |   maxOccurs="1" type="xsd:int"/>
|                      |   <xsd:element name="y" minOccurs="1"
|                      |   maxOccurs="1" type="xsd:int"/>
|                      | </xsd:sequence>
|                      | </xsd:complexType> |
| <!-- @struct true --> | <!-- @language_binding flat-data--> |
| <!-- @transfer_mode shmem_ref --> | <!-- @extensibility final--> |

---

3.5.1 Primitive Types

The primitive types char, wchar, long double, and wstring are not supported natively in XSD. Connext DDS provides definitions for these types in the file `<NDDSHOME>/resource/app/app_
support/rtiddgen/schema. All files that use the primitive types char, wchar, long double and wstring must reference rti.dds_topic_types_common.xsd. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:dds="http://www.omg.org/dds">
  <xsd:import namespace="http://www.omg.org/dds"
    schemaLocation="rti.dds_topic_types_common.xsd"/>
  <xsd:complexType name="Foo">
    <xsd:sequence>
      <xsd:element name="myChar" minOccurs="1"
        maxOccurs="1" type="dds:char"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

### 3.6 Using RTI Code Generator (rtiddgen)

*RTI Code Generator* creates the code needed to define and register a user-data type with *Connext DDS*. Using this tool is optional if:

- You are using dynamic types (see 3.8 Interacting Dynamically with User Data Types on page 154)
- You are using one of the built-in types (see 3.2 Built-in Data Types on page 38)

See the [RTI Code Generator User's Manual](#) for more information.

### 3.7 Using Generated Types without Connext DDS (Standalone)

You can use the generated type-specific source and header files without linking the *Connext DDS* libraries or even including the *Connext DDS* header files. That is, the files generated by *RTI Code Generator* for your data types can be used standalone.

The directory `<NDDSHOME>/resource/app/app_support/rtiddgen/standalone` contains the required helper files:

- include: header and templates files for C and C++.
- src: source files for C and C++.
- class: Java jar file.

#### 3.7.1 Using Standalone Types in C

The generated files that can be used standalone are:
3.7.2 Using Standalone Types in C++

- `<idl file name>.c`: Types source file
- `<idl file name>.h`: Types header file

The type plug-in code (<idl file>Plugin.[c,h]) and type-support code (<idl file>Support.[c,h]) cannot be used standalone.

To use the generated types in a standalone manner:

1. Include the directory `<NDDSHOME>/resource/app/app_support/rtiddsgen/standalone/include` in the list of directories to be searched for header files.
2. Add the source files, `ndds_standalone_type.c` and `<idl file name>.c`, to your project.
3. Include the file `<idl file name>.h` in the source files that will use the generated types in a standalone manner.
4. Compile the project using the following two preprocessor definitions:
   - `NDDS_STANDALONE_TYPE`
   - The definition for your platform (RTI_VXWORKS, RTI_QNX, RTI_WIN32, RTI_INTY, RTI_LYNX or RTI_UNIX)

3.7.2 Using Standalone Types in C++

(This section applies to the Traditional C++ API only)

The generated files that can be used standalone are:

- `<idl file name>.cxx`: Types source file
- `<idl file name>.h`: Types header file

The type-plugin code (<idl file>Plugin.[cxx,h]) and type-support code (<idl file>Support.[cxx,h]) cannot be used standalone.

To use the generated types in a standalone manner:

1. Include the directory `<NDDSHOME>/resource/app/app_support/rtiddsgen/standalone/include` in the list of directories to be searched for header files.
2. Add the source files, `ndds_standalone_type.cxx` and `<idl file name>.cxx`, to your project.
3. Include the file `<idl file name>.h` in the source files that will use the RTI Code Generator types in a standalone manner.
4. Compile the project using the following two preprocessor definitions:
   - NDDS_STANDALONE_TYPE
   - The definition for your platform (such as RTI_VXWORKS, RTI_QNX, RTI_WIN32, RTI_INTY, RTI_LYNX or RTI_UNIX)

3.7.3 Standalone Types in Java

The generated files that can be used standalone are:

- `<idl type>.java`
- `<idl type>Seq.java`

The type code (`<idl file>TypeCode.java`), type-support code (`<idl file>TypeSupport.java`), `DataReader` code (`<idl file>DataReader.java`) and `DataWriter` code (`<idl file>DataWriter.java`) cannot be used standalone.

To use the generated types in a standalone manner:

1. Include the file `ndds_standalone_type.jar` in the classpath of your project.
2. Compile the project using the standalone types files (`<idl type>.java` and `<idl type>Seq.java`).

3.8 Interacting Dynamically with User Data Types

3.8.1 Type Schemas and TypeCode Objects

Type schemas—the names and definitions of a type and its fields—are represented by TypeCode objects, described in 3.1.3 Introduction to TypeCode on page 37.

3.8.2 Defining New Types

This section does not apply when using the separate add-on product, Ada Language Support, which does not support Dynamic Types.

Locally, your application can access the type code for a generated type "Foo" by calling the `FooTypeSupport::get_typecode()` (Traditional C++ Notation) operation in the code for the type generated by RTI Code Generator. But you can also create TypeCodes at run time without any code generation.

Creating a TypeCode is parallel to the way you would define the type statically: you define the type itself with some name, then you add members to it, each with its own name and type.

For example, consider the following statically defined type. It might be in C, C++, or IDL; the syntax is largely the same.
struct MyType {
    int32 my_integer;
    float my_float;
    bool my_bool;
    @key string<128> my_string;
};

This is how you would define the same type at run time in the Traditional C++ API:

```cpp
DDS_ExceptionCode_t ex = DDS_NO_EXCEPTION_CODE;
DDS_StructMemberSeq structMembers; // ignore for now
DDS_TypeCodeFactory* factory =
    DDS_TypeCodeFactory::get_instance();
DDS_TypeCode* structTc = factory->create_struct_tc(
    "MyType", structMembers, ex);
// If structTc is NULL, check 'ex' for more information.
structTc->add_member(
    "my_integer", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_LONG),
    DDS_TYPECODE_NONKEY_REQUIRED_MEMBER, ex);
structTc->add_member(
    "my_float", DDS_TYPECODE.Member_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_FLOAT),
    DDS_TYPECODE_NONKEY_REQUIRED_MEMBER, ex);
structTc->add_member(
    "my_bool", DDS_TYPECODE_MEMBER_ID_INVALID,
    factory->get_primitive_tc(DDS_TK_BOOLEAN),
    DDS_TYPECODE_NONKEY_REQUIRED_MEMBER, ex);
structTc->add_member(
    "my_string", DDS_TYPECODE.Member_ID_INVALID,
    factory->create_string_tc(128),
    DDS_TYPECODE.KEY_MEMBER, ex);
```

More detailed documentation for the methods and constants you see above, including example code, can be found in the API Reference HTML documentation, which is available for all supported programming languages.

If, as in the example above, you know all of the fields that will exist in the type at the time of its construction, you can use the `StructMemberSeq` to simplify the code:
### 3.8.3 Sending Only a Few Fields

In some cases, your data model may contain a large number of potential fields, but it may not be desirable or appropriate to include a value for every one of them with every DDS data sample.

- **It may use too much bandwidth.** You may have a very large data structure, parts of which are updated very frequently. Rather than resending the entire data structure with every change, you may wish to send only those fields that have changed and rely on the recipients to reassemble the complete state themselves.

- **It may not make sense.** Some fields may only have meaning in the presence of other fields. For example, you may have an event stream in which certain fields are only relevant for certain kinds of events.

To support these and similar cases, Connext DDS supports mutable types and optional members (see the RTI Connext DDS Core Libraries Extensible Types Guide).
3.8.4 Sending Type Information on the Network

In addition to being used locally, the type information of a Topic is published automatically during discovery as part of the builtin topics for publications and subscriptions. See 3.1.3.1 Sending Type Information on the Network on page 38.

3.8.4.1 Type Codes for Built-in Types

The type codes associated with the built-in types are generated from the following IDL type definitions:

```idl
module DDS {
    /* String */
    struct String {
        string<max_size> value;
    };
    /* KeyedString */
    struct KeyedString {
        string<max_size> key; // @key
        string<max_size> value;
    };
    /* Octets */
    struct Octets {
        sequence<octet, max_size> value;
    };
    /* KeyedOctets */
    struct KeyedOctets {
        string<max_size> key; // @key
        sequence<octet, max_size> value;
    };
};
```

The maximum size (**max_size**) of the strings and sequences that will be included in the type code definitions can be configured on a per-**DomainParticipant**-basis by using the properties in Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant.

<table>
<thead>
<tr>
<th>Built-in Type</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>dds.builtin_type.string.max_size</td>
<td>Maximum size of the strings published by the DataWriters and received by the DataReaders belonging to a DomainParticipant (includes the NULL-terminated character). Default: 1024</td>
</tr>
</tbody>
</table>
Table 3.14 Properties for Allocating Size of Built-in Types, per DomainParticipant

<table>
<thead>
<tr>
<th>Built-in Type</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KeyedString</td>
<td>dds.builtin_type.keyed_string.max_key_size</td>
<td>Maximum size of the keys used by the DataWriters and DataReaders belonging to a DomainParticipant (includes the NULL-terminated character). Default: 1024</td>
</tr>
<tr>
<td>Octets</td>
<td>dds.builtin_type.octets.max_size</td>
<td>Maximum size of the octet sequences published by the DataWriters and DataReaders belonging to a DomainParticipant. Default: 2048</td>
</tr>
<tr>
<td>Keyed-Octets</td>
<td>dds.builtin_type.keyed_octets.max_key_size</td>
<td>Maximum size of the key published by the DataWriter and received by the DataReaders belonging to the DomainParticipant (includes the NULL-terminated character). Default: 1024.</td>
</tr>
<tr>
<td></td>
<td>dds.builtin_type.keyed_octets.max_size</td>
<td>Maximum size of the octet sequences published by the DataWriters and DataReaders belonging to a DomainParticipant. Default: 2048</td>
</tr>
</tbody>
</table>

3.9 Working with DDS Data Samples

You should now understand how to define and work with data types, whether you're using the simple data types built into the middleware (see 3.2 Built-in Data Types on page 38), dynamically defined types (see 3.2.7 Managing Memory for Built-in Types on page 69), or code generated from IDL or XML files (see 3.3 Creating User Data Types with IDL on page 75 and 3.4 Creating User Data Types with Extensible Markup Language (XML) on page 121).

Now that you have chosen one or more data types to work with, this section will help you understand how to create and manipulate objects of those types.

3.9.1 Objects of Concrete Types

If you use one of the built-in types or decide to generate custom types from an IDL or XML file, your Connext DDS data type is like any other data type in your application: a class or structure with fields, methods, and other members that you interact with directly.

In C

You create and delete your own objects from factories, just as you create Connext DDS objects from factories. In the case of user data types, the factory is a singleton object called the type support. Objects allocated from these factories are deeply allocated and fully initialized.
3.9.1 Objects of Concrete Types

```
/* In the generated header file: */
struct MyData {
    char* myString;
};
/* In your code: */
MyData* sample = MyDataTypeSupport_create_data();
char* str = sample->myString; /*empty, non-NULL string*/
/* ... */
MyDataTypeSupport_delete_data(sample);
```

**In Traditional C++:**

Without the `-constructor` option, you create and delete objects using the TypeSupport factories.

```
MyData* sample = MyDataTypeSupport::create_data();
char* str = sample->myString; // empty, non-NULL string
// ...
MyDataTypeSupport::delete_data(sample);
```

With the `-constructor` option, generated types have a default constructor, a copy constructor, and a destructor. In this case the TypeSupport data creation methods are not available.

```
// In the header file
class MyType
{
    MyType();
    MyType(const MyType& that);
    ~MyType();
    MyType& operator=(const MyType& that);
};
```

**In Modern C++:**

Generated types have value-type semantics and provide a default constructor, a constructor with parameters to initialize all the members, a copy constructor and assignment operator, a move constructor and move-assignment operator, a destructor, equality operators, a swap function and an overloaded operator `<<`. Data members are accessed using getters and setters.

```
// In the generated header file
class MyData {
public:
    MyData();
    explicit MyData(const std::string& myString);

    // Note: the implicit destructor, copy and move constructors, and assignment operators
    // are available
    std::string& myString() OMG_NOEXCEPT;
    const std::string& myString() const OMG_NOEXCEPT;
    void myString(const std::string& value);
```
3.9.1 Objects of Concrete Types

```cpp
bool operator == (const MyData& other_) const;
bool operator != (const MyData& other_) const;
private:
    // ...
};

void swap(MyData& a, MyData& b) OMG_NORETURN
    std::ostream& operator <<
    (std::ostream& o,const MyData& sample);

// In your code:
MyData sample("Hello");
sample.myString("Bye");
```

**In C# and C++/CLI:**

You can use a no-argument constructor to allocate objects. Those objects will be deallocated by the garbage collector as appropriate.

```csharp
// In the generated code (C++/CLI):
public ref struct MyData {
    public: System::String^ myString;
};

// In your code, if you are using C#:
MyData sample = new MyData();
System.String str = sample.myString;
// empty, non-null string

// In your code, if you are using C++/CLI:
MyData^ sample = gcnew MyData();
System::String^ str = sample->myString;
// empty, non-nullptr string
```

**In Java:**

You can use a no-argument constructor to allocate objects. Those objects will be deallocated by the garbage collector as appropriate.

```java
// In the generated code:
public class MyData {
    public String myString = "";
}
// In your code:
MyData sample = new MyData();
```
3.9.2 Objects of Dynamically Defined Types

If you are working with a data type that was discovered or defined at run time, you will use the reflective API provided by the DynamicData class to get and set the fields of your object.

Consider the following type definition:

```c
struct MyData {
    int32 myInteger;
};
```

As with a statically defined type, you will create objects from a TypeSupport factory. How to create or otherwise obtain a TypeCode, and how to subsequently create from it a DynamicDataTypeSupport, is described in 3.8.2 Defining New Types on page 154. In the Modern C++ API you will use the DynamicData constructor, which receives a DynamicType.

For more information about the DynamicData and DynamicDataTypeSupport classes, consult the API Reference HTML documentation, which is available for all supported programming languages (select Modules, RTI Connext DDS API Reference, Topic Module, Dynamic Data).

In C:

```c
DDS_DynamicDataTypeSupport* support = ...;
DDS_DynamicData* sample = DDS_DynamicDataTypeSupport_create_data(support);
DDS_Long theInteger = 0;
DDS_ReturnCode_t success = DDS_DynamicData_set_long(sample,
    "myInteger", DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, 5);
/* Error handling omitted. */
success = DDS_DynamicData_get_long( sample, &theInteger,
    "myInteger", DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
/* Error handling omitted. "theInteger" now contains the value 5
    if no error occurred. */
```

In Traditional C++:

```c
DDS_DynamicDataTypeSupport* support = ...;
DDS_DynamicData* sample = support->create_data();
DDS_ReturnCode_t success = sample->set_long("myInteger",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED, 5);
// Error handling omitted.
DDS_Long theInteger = 0;
success = sample->get_long( &theInteger, "myInteger",
    DDS_DYNAMIC_DATA_MEMBER_ID_UNSPECIFIED);
// Error handling omitted.
// "theInteger" now contains the value 5 if no error occurred.
```

In Modern C++:

```c
using namespace dds::core::xtypes;
```
Objects of Dynamically Defined Types

StructType type(
    "MyData", {
        Member("myInteger", primitive_type<int32_t>())
    })
DynamicData sample(type);
sample.value("myInteger", 5);
int32_t the_int = sample.value<int32_t>("myInteger");
// "the_int" now contains the value 5 if no exception was thrown

In C++/CLI:

using DDS;
DynamicDataTypeSupport^ support = ...;
DynamicData^ sample = support->create_data();
sample->set_long("myInteger",
    DynamicData::MEMBER_ID_UNSPECIFIED, 5);
int theInteger = sample->get_long("myInteger",
    0 /*redundant w/ field name*/);
/* Exception handling omitted.
   "theInteger" now contains the value 5 if no error occurred.
*/

In C#:

using namespace DDS;
DynamicDataTypeSupport support = ...;
DynamicData sample = support.create_data();
sample.set_long("myInteger", DynamicData.MEMBER_ID_UNSPECIFIED, 5);
int theInteger = sample.get_long("myInteger",
    DynamicData.MEMBER_ID_UNSPECIFIED);
/* Exception handling omitted.
   "theInteger" now contains the value 5 if no error occurred.
*/

In Java:

import com.rti.dds.dynamicdata.*;
DynamicDataTypeSupport support = ...;
DynamicData sample = (DynamicData) support.create_data();
sample.set_int("myInteger", DynamicData.MEMBER_ID_UNSPECIFIED, 5);
int theInteger = sample.get_int("myInteger",
    DynamicData.MEMBER_ID_UNSPECIFIED);
/* Exception handling omitted.
   "theInteger" now contains the value 5 if no error occurred.
*/

The Modern C++ API provides convenience functions to convert among DynamicData samples and typed samples (such as MyData, from the previous example). For example:

#include "MyData.hpp"
// ...
MyData typed_sample(44);
DynamicData dynamic_sample = rti::core::xtypes::convert(typed_sample);
assert (dynamic_sample.value<int32_t>("myInteger") == 44);
3.9.3 Serializing and Deserializing Data Samples

There are two TypePlugin operations to serialize a sample into a buffer and deserialize a sample from a buffer. The sample serialization/deserialization uses CDR representation.

The feature is supported in the following languages: C, Modern and Traditional C++, Java, and .NET.

C:

```c
#include "FooSupport.h"

FooTypeSupport::serialize_data_to_cdr_buffer(...)
FooTypeSupport::deserialize_data_from_cdr_buffer(...)```

Traditional C++

```cpp
#include "FooSupport.h"

FooTypeSupport::serialize_data_to_cdr_buffer(...)
FooTypeSupport::deserialize_data_from_cdr_buffer(...)```

Modern C++

```cpp
#include "Foo.hpp"

dds::topic::topic_type_support<Foo>::to_cdr_buffer(...)  
dds::topic::topic_type_support<Foo>::from_cdr_buffer(...)```

Java:

```java
FooTypeSupport.get_instance().serialize_to_cdr_buffer(...)  
FooTypeSupport.get_instance().deserialize_from_cdr_buffer(...)```

C++/CLI:

```csharp
FooTypeSupport::serialize_data_to_cdr_buffer(...)
FooTypeSupport::deserialize_data_from_cdr_buffer(...)```

C#:

```csharp
FooTypeSupport::serialize_data_to_cdr_buffer(...)
FooTypeSupport::deserialize_data_from_cdr_buffer(...)```

3.9.4 Accessing the Discriminator Value in a Union

A union type can only hold a single member. The member_id for this member is equal to the discriminator value. To get the value of the discriminator, use the operation get_member_info_by_index() on the DynamicData using an index value of 0. This operation fills in a DynamicDataMemberInfo structure, which includes a member_id field that is the value of the discriminator.

Once you know the discriminator value, you can use the proper version of get_<type>() (such as get_long()) to access the member value.

For example:

dynamic_sample.value("myInteger", 33);
typed_sample = rti::core::xtypes::convert<MyData>(dynamic_sample);
assert (typed_sample.myInteger() == 33);
DynamicDataMemberInfo memberInfo = new DynamicDataMemberInfo();
myDynamicData.get_member_info_by_index(memberInfo, 0);
int discriminatorValue = memberInfo.member_id;
int myMemberValue = myDynamicData.get_long(null, discriminatorValue);

The Modern C++ API provides the method `discriminator_value()` to achieve the same result:

```cpp
int32_t my_member_value = my_dynamic_data.value<int32_t>(
    my_dynamic_data.discriminator_value());
```

### 3.10 Data Sample Serialization Limits

`Connext DDS` does not support data types with samples whose maximum serialized size is bigger than 2,147,482,623 bytes. If the maximum serialized size is bigger than this, the behavior is undefined.

For types containing unbounded sequences or strings where the code is generated using the `-unboundedSupport` command-line option in `RTI Code Generator`, the maximum serialized size is implicitly set to 2,147,482,623.
Chapter 4 DDS Entities

The main classes extend an abstract base class called a DDS Entity. Every DDS Entity has a set of associated events known as statuses and a set of associated Quality of Service Policies (QosPolicies). In addition, a Listener may be registered with the Entity to be called when status changes occur. DDS Entities may also have attached DDS Conditions, which provide a way to wait for status changes. Figure 4.1: Overview of DDS Entities on the next page presents an overview in a UML diagram.

This section describes the common operations and general designed patterns shared by all DDS Entities including DomainParticipants, Topics, Publishers, DataWriters, Subscribers, and DataReaders. In subsequent chapters, the specific statuses, Listeners, Conditions, and QosPolicies for each class will be discussed in detail.
4.1 Common Operations for All DDS Entities

All DDS Entities (DomainParticipants, Topics, Publishers, DataWriters, Subscribers, and DataReaders) provide operations for:
4.1.1 Creating and Deleting DDS Entities

- C, Traditional C++, Java, and .NET:

The factory design pattern is used in creating and deleting DDS Entities. Instead of declaring and constructing or destructing Entities directly, a factory object is used to create an Entity. Almost all Entity factories are objects that are also Entities. The only exception is the factory for a DomainParticipant. See Table 4.1 Entity Factories.

<table>
<thead>
<tr>
<th>Entity</th>
<th>Created by</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainParticipant</td>
<td>DomainParticipantFactory (a static singleton object provided by Connext DDS)</td>
</tr>
<tr>
<td>Topic</td>
<td></td>
</tr>
<tr>
<td>Publisher</td>
<td>DomainParticipant</td>
</tr>
<tr>
<td>Subscriber</td>
<td>DomainParticipant</td>
</tr>
<tr>
<td>DataWriter</td>
<td>Publisher</td>
</tr>
<tr>
<td>DataReader</td>
<td>Subscriber</td>
</tr>
</tbody>
</table>

All Entities that are factories have:
- Operations to create and delete child Entities. For example:

  ```cpp
  DDSPublisher::create_datawriter()
  DDSDomainParticipant::delete_topic()
  ```

- Operations to get and set the default QoS values used when creating child Entities. For example:

  ```cpp
  DDSSubscriber::get_default_datareader_qos()
  DDSDomainParticipantFactory::set_default_participant_qos()
  ```

And 7.4.2 ENTITYFACTORY QosPolicy on page 348 to specify whether or not the newly created child Entity should be automatically enabled upon creation.

DataWriters may be created by a DomainParticipant or a Publisher. Similarly, DataReaders may be created by a DomainParticipant or a Subscriber.

An entity that is a factory cannot be deleted until all the child Entities created by it have been deleted.
Each *Entity* obtained through `create_<entity>()` must eventually be deleted by calling `delete_<entity>()`, or by calling `delete_contained_entities()`.

- Modern C++:

In the Modern C++ API the factory pattern is not explicit. Entities have constructors and destructors. The first argument to an Entity's constructor is its "factory" (except for the DomainParticipant). For example:

```cpp
// Note: this example shows the simplest version of each Entity's constructor:
dds::domain::DomainParticipant participant(MY_DOMAIN_ID);
dds::topic::Topic<Foo> topic(participant, "Example Foo");
dds::sub::Subscriber subscriber(participant);
dds::sub::DataReader<Foo> reader(subscriber, topic);
dds::pub::Publisher publisher(participant);
dds::pub::DataWriter<Foo> writer(publisher, topic);
```

Entities are reference types. In a reference type copy operations, such as copy-construction and copy-assignment are shallow. The reference types are modeled after shared pointers. Similar to pointers, it is important to distinguish between an entity and a reference (or handle) to it. A single entity may have multiple references. Copying a reference does not copy the entity it is referring to—creating additional references from the existing reference(s) is a relatively inexpensive operation.

The lifecycle of references and the entity they are referring to is not the same. In general, the entity lives as long as there is at least one reference to it. When the last reference to the entity ceases to exist, the entity it is referring to is destroyed.

Applications can override the automatic destruction of Entities. An Entity can be explicitly closed (by calling the method `close()` or retained (by calling `retain()`)

Closing an Entity destroys the underlying object and invalidates all references to it.

Retaining an Entity disables the automatic destruction when it loses all its reference. A retained Entity can be looked up (see 9.2.4 Looking Up DomainParticipants on page 608) and has to be explicitly destroyed with `close()`.

### 4.1.2 Enabling DDS Entities

The `enable()` operation changes an *Entity* from a non-operational to an operational state. *Entity* objects can be created disabled or enabled. This is controlled by the value of the 7.4.2 ENTITYFACTORY QosPolicy on page 348 on the corresponding factory for the *Entity* (not on the *Entity* itself).

By default, all *Entities* are automatically created in the enabled state. This means that as soon as the *Entity* is created, it is ready to be used. In some cases, you may want to create the *Entity* in a ‘disabled’ state. For example, by default, as soon as you create a `DataReader`, the `DataReader` will start receiving new DDS samples for its `Topic` if they are being sent. However, your application may still be initializing other components and may not be ready to process the data at that time. In that case, you can tell the `Subscriber` to
create the DataReader in a disabled state. After all of the other parts of the application have been created and initialized, then the DataReader can be enabled to actually receive messages.

To create a particular entity in a disabled state, modify the EntityFactory QosPolicy of its corresponding factory entity before calling create_{<entity>}. For example, to create a disabled DataReader, modify the Subscriber’s QoS as follows:

```c
DDS_SubscriberQos subscriber_qos;
subscriber->get_qos(subscriber_qos);
subscriber_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
subscriber->set_qos(subscriber_qos);
DDSDataReader* datareader =
    subscriber->create_datareader(topic, DDS_DATAREADER_QOS_DEFAULT, listener);
```

When the application is ready to process received data, it can enable the DataReader:

```c
datareader->enable();
```

### 4.1.2.1 Rules for Calling enable()

In the following, a ‘Factory’ refers to a DomainParticipant, Publisher, or Subscriber; a ‘child’ refers to an entity created by the factory:

- If the factory is disabled, its children are always created disabled.
- If the factory is enabled, its children will be created according to the setting in the factory’s EntityFactoryQoS value.
- Calling enable() on a child whose factory object is still disabled will fail and return DDS_RECODE_RECONDITION_NOT_MET.
- Calling enable() on a factory with EntityFactoryQoS set to DDS_BOOLEAN_TRUE will recursively enable all of the factory’s children. If the factory’s EntityFactoryQoS is set to DDS_BOOLEAN_FALSE, only the factory itself will be enabled.
- Calling enable() on an entity that is already enabled returns DDS_RET CODE_OK and has no effect.
- There is no complementary “disable” operation. You cannot disable an entity after it is enabled. Disabled Entities must have been created in that state.
- An entity’s Listener will only be invoked if the entity is enabled.
- The existence of an entity is not propagated to other DomainParticipants until the entity is enabled (see Discovery (Chapter 15 on page 764)).
- If a DataWriter/DataReader is to be created in an enabled state, then the associated Topic must already be enabled. The enabled state of the Topic does not matter, if the Publisher/Subscriber has its EntityFactory QosPolicy to create children in a disabled state.
When calling enable() for a DataWriter/DataReader, both the Publisher/Subscriber and the Topic must be enabled, or the operation will fail and return DDS RETCODE PRECONDITION_NOT_MET.

The following operations may be invoked on disabled Entities:

- **get_qos()** and **set_qos()** Some DDS-specified QosPolicies are immutable—they cannot be changed after an Entity is enabled. This means that for those policies, if the entity was created in the disabled state, get/set_qos() can be used to change the values of those policies until enabled() is called on the Entity. After the Entity is enabled, changing the values of those policies will not affect the Entity. However, there are mutable QosPolicies whose values can be changed at anytime—even after the Entity has been enabled.

Finally, there are extended QosPolicies that are not a part of the DDS specification but offered by Connext DDS to control extended features for an Entity. Some of those extended QosPolicies cannot be changed after the Entity has been created—regardless of whether the Entity is enabled or disabled.

Into which exact categories a QosPolicy falls—mutable at any time, immutable after enable, immutable after creation—is described in the documentation for the specific policy.

- **get_status_changes()** and **get_*_status()** The status of an Entity can be retrieved at any time (but the status of a disabled Entity never changes). (Note: get_*_status() resets the related status so it no longer considered “changed.”)

- **get_statuscondition()** An Entity’s StatusCondition can be checked at any time (although the status of a disabled Entity never changes).

- **get_listener()** and **set_listener()** An Entity’s Listener can be changed at any time.

- **create_*()** and **delete_*()** A factory Entity can still be used to create or delete any child Entity that it can produce. Note: following the rules discussed previously, a disabled Entity will always create its children in a disabled state, no matter what the value of the EntityFactory QosPolicy is.

- **lookup_*()** An Entity can always look up children it has previously created.

Most other operations are not allowed on disabled Entities. Executing one of those operations when an Entity is disabled will result in a return code of DDS RETCODE NOT_ENABLED. The documentation for a particular operation will explicitly state if it is not allowed to be used if the Entity is disabled.

The built-in transports are implicitly registered when (a) the DomainParticipant is enabled, (b) the first DataWriter/DataReader is created, or (c) you look up a built-in data reader, whichever happens first. Any changes to the built-in transport properties that are made after the built-in transports have been registered will have no affect on any DataWriters/DataReaders.
4.1.3 Getting an Entity’s Instance Handle

The *Entity* class provides an operation to retrieve an instance handle for the object. The operation is simply:

```
InstanceHandle_t get_instance_handle()
```

An instance handle is a global ID for the entity that can be used in methods that allow user applications to determine if the entity was locally created, if an entity is owned (created) by another entity, etc.

4.1.4 Getting Status and Status Changes

The `get_status_changes()` operation retrieves the set of events, also known in DDS terminology as *communication statuses*, in the *Entity* that have changed since the last time `get_status_changes()` was called. This method actually returns a value that must be bitwise AND’ed with an enumerated bit mask to test whether or not a specific status has changed. The operation can be used in a polling mechanism to see if any statuses related to the *Entity* have changed. If an entity is disabled, all communication statuses are in the “unchanged” state so the list returned by the `get_status_changes()` operation will be empty.

A set of statuses is defined for each class of *Entities*. For each status, there is a corresponding operation, `get_<status-name>_status()`, that can be used to get its current value. For example, a *DataWriter* has a `DDS_OFFERED_DEADLINE_MISSED` status; it also has a `get_offered_deadline_missed_status()` operation:

```
DDS_StatusMask statuses;
DDS_OfferedDeadlineMissedStatus deadline_stat;
statuses = datawriter->get_status_changes();
if (statuses & DDS_OFFERED_DEADLINE_MISSED_STATUS) {
    datawriter->get_offered_deadline_missed_status(
        &deadline_stat);
    printf("Deadline missed %d times.\n",
           deadline_stat.total_count);
}
```

To reset a status (so that it is no longer considered “changed”), call `get_<status-name>_status()`. Or, in the case of the `DDS_DATA_AVAILABLE` status, call `read()`, `take()`, or one of their variants.

> If you use a *StatusCondition* to be notified that a particular status has changed, the *StatusCondition*’s `trigger_value` will remain true unless you call `get*_status()` to reset the status.

See also: 4.3 Statuses on page 183 and 4.6.8 StatusConditions on page 212.

4.1.5 Getting and Setting Listeners

Each type of *Entity* has an associated *Listener*, see 4.4 Listeners on page 189. A *Listener* represents a set of functions that users may install to be called asynchronously when the state of *communication statuses* change.

The `get_listener()` operation returns the current *Listener* attached to the *Entity*. 
The **set_listener**() operation installs a **Listener** on an **Entity**. The **Listener** will only be invoked on the changes of statuses specified by the accompanying mask. Only one listener can be attached to each **Entity**. If a **Listener** was already attached, **set_listener**() will replace it with the new one.

The **get_listener**() and **set_listener**() operations are directly provided by the **DomainParticipant**, **Topic**, **Publisher**, **DataWriter**, **Subscriber**, and **DataReader** classes so that listeners and masks used in the argument list are specific to each **Entity**.

**Note:** The **set_listener**() operation is not synchronized with the listener callbacks, so it is possible to set a new listener on an participant while the old listener is in a callback. Therefore you should be careful not to delete any listener that has been set on an enabled participant unless some application-specific means are available of ensuring that the old listener cannot still be in use.

See 4.4 **Listeners on page 189** for more information about **Listeners**.

### 4.1.6 Getting the StatusCondition

Each type of **Entity** may have an attached **StatusCondition**, which can be accessed through the **get_statuscondition**() operation. You can attach the **StatusCondition** to a **WaitSet**, to cause your application to wait for specific status changes that affect the **Entity**.

See 4.6 **Conditions and WaitSets on page 202** for more information about **StatusConditions** and **WaitSets**.

### 4.1.7 Getting, Setting, and Comparing QosPolicies

Each type of **Entity** has an associated set of QosPolicies (see 4.2 **QosPolicies on page 176**). QosPolicies allow you to configure and set properties for the **Entity**.

While most QosPolicies are defined by the DDS specification, some are offered by Connext DDS as extensions to control parameters specific to the implementation.

There are two ways to specify a QoS policy:

- Programmatically, as described in this section.
- QosPolicies can also be configured from XML resources (files, strings)—with this approach, you can change the QoS without recompiling the application. The QoS settings are automatically loaded by the DomainParticipantFactory when the first **DomainParticipant** is created. See **Configuring QoS with XML** (Chapter 19 on page 848).

The **get_qos**() operation retrieves the current values for the set of QosPolicies defined for the **Entity**.

QosPolicies can be set programmatically when an **Entity** is created, or modified with the **Entity**'s **set_qos**() operation.

The **set_qos**() operation sets the QosPolicies of the entity. Note: not all QosPolicy changes will take effect instantaneously; there may be a delay since some QosPolicies set for one entity, for example, a
DataWriter, may actually affect the operation of a matched entity in another application, for example, a DataWriter.

The get_qos() and set_qos() operations are passed QoS structures that are specific to each derived entity class, since the set of QosPolicies that effect each class of Entities is different.

The equals() operation compares two Entity’s QoS structures for equality. It takes two parameters for the two Entities’ QoS structures to be compared, then returns TRUE is they are equal (all values are the same) or FALSE if they are not equal.

Each QosPolicy has default values (listed in the API Reference HTML documentation). If you want to use custom values, there are three ways to change QosPolicy settings:

- Before Entity creation (if custom values should be used for multiple Entities). See 4.1.7.1 Changing the QoS Defaults Used to Create DDS Entities: set_default_*_qos() below.
- During Entity creation (if custom values are only needed for a particular Entity). See 4.1.7.2 Setting QoS During Entity Creation on the next page.
- After Entity creation (if the values initially specified for a particular Entity are no longer appropriate). See 4.1.7.3 Changing the QoS for an Existing Entity on page 175.

Regardless of when or how you make QoS changes, there are some rules to follow:

- Some QosPolicies interact with each other and thus must be set in a consistent manner. For instance, the maximum value of the HISTORY QosPolicy’s depth parameter is limited by values set in the RESOURCE_LIMITS QosPolicy. If the values within a QosPolicy structure are inconsistent, then set_qos() will return the error INCONSISTENT_POLICY, and the operation will have no effect.
- Some policies can only be set when the Entity is created, or before the Entity is enabled. Others can be changed at any time. In general, all standard DDS QosPolicies can be changed before the Entity is enabled. A subset can be changed after the Entity is enabled. Connext DDS-specific QosPolicies either cannot be changed after creation or can be changed at any time. The changeability of each QosPolicy is documented in the API Reference HTML documentation as well as in Table 4.2 QosPolicies. If you attempt to change a policy after it cannot be changed, set_qos() will fail with a return IMMUTABLE_POLICY.

### 4.1.7.1 Changing the QoS Defaults Used to Create DDS Entities: set_default_*_qos()

Each parent factory has a set of default QoS settings that are used when the child entity is created. The DomainParticipantFactory has default QoS values for creating DomainParticipants. A DomainParticipant has a set of default QoS for each type of entity that can be created from the DomainParticipant (Topic, Publisher, Subscriber, DataWriter, and DataReader). Likewise, a Publisher has a set of default QoS values used when creating DataWriters, and a Subscriber has a set of default QoS values used when creating DataReaders.
An entity’s QoS are set when it is created. Once an entity is created, all of its QoS—for itself and its child Entities—are fixed unless you call set_qos() or set_qos_with_profile() on that entity. Calling set_default_<entity>_qos() on a parent entity will have no effect on child Entities that have already been created.

You can change these default values so that they are automatically applied when new child Entities are created. For example, suppose you want all DataWriters for a particular Publisher to have their RELIABILITY QosPolicy set to RELIABLE. Instead of making this change for each DataWriter when it is created, you can change the default used when any DataWriter is created from the Publisher by using the Publisher’s set_default_datawriter_qos() operation.

```c
DDS_DataWriterQos default_datawriter_qos;
// get the current default values
publisher->get_default_datawriter_qos(default_datawriter_qos);
// change to desired default values
default_datawriter_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
// set the new default values
publisher->set_default_datawriter_qos(default_datawriter_qos);
// created datawriters will use new default values
datawriter =
    publisher->create_datawriter(topic, NULL, NULL, NULL);
```

It is not safe to get or set the default QoS values for an entity while another thread may be simultaneously calling get_default_<entity>_qos(), set_default_<entity>_qos(), or create_<entity>() with DDS_<ENTITY>_QOS_DEFAULT as the qos parameter (for the same entity).

Another way to make QoS changes is by using XML resources (files, strings). For more information, see Configuring QoS with XML (Chapter 19 on page 848).

### 4.1.7.2 Setting QoS During Entity Creation

If you only want to change a QosPolicy for a particular entity, you can pass in the desired QosPolicies for an entity in its creation routine.

To customize an entity's QoS before creating it:

1. (C API Only) Initialize a QoS object with the appropriate INITIALIZER constructor.
2. Call the relevant get_<entity>_default_qos() method.
3. Modify the QoS values as desired.
4. Create the entity.
4.1.7 Getting, Setting, and Comparing QosPolicies

For example, to change the RELIABLE QosPolicy for a DataWriter before creating it:

```c++
// Initialize the QoS object
DDS_DataWriterQos datawriter_qos;
// Get the default values
publisher->get_default_datawriter_qos(datawriter_qos);
// Modify the QoS values as desired
datawriter_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
// Create the DataWriter with new values
datawriter = publisher->create_datawriter(
    topic, datawriter_qos, NULL, NULL);
```

Another way to set QoS during entity creation is by using a QoS profile. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

4.1.7.3 Changing the QoS for an Existing Entity

Some policies can also be changed after the entity has been created. To change such a policy after the entity has been created, use the entity’s set_qos() operation.

For example, suppose you want to tweak the DEADLINE QoS for an existing DataWriter:

```c++
DDS_DataWriterQos datawriter_qos;
// get the current values
datawriter->get_qos(datawriter_qos);
// make desired changes
datawriter_qos.deadline.period.sec = 3;
datawriter_qos.deadline.period.nanosec = 0;
// set new values
datawriter->set_qos(datawriter_qos);
```

Another way to make QoS changes is by using a QoS profile. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

**Note:** In the code examples presented in this section, we are not testing for the return code for the set_qos(), set_default_*_qos() functions. If the values used in the QosPolicy structures are inconsistent then the functions will fail and return INCONSISTENT_POLICY. In addition, set_qos() may return IMMUTABLE_POLICY if you try to change a QosPolicy on an Entity after that policy has become immutable. *User code should test for and address those anomalous conditions.*

4.1.7.4 Default QoS Values

Connext DDS provides special constants for each Entity type that can be used in set_qos() and set_default_*_qos() to reset the QosPolicy values to the original DDS default values:

- DDS_PARTICIPANT_QOS_DEFAULT
- DDS PUBLISHER_QOS_DEFAULT
- DDS_SUBSCRIBER_QOS_DEFAULT
4.2 QosPolicies

- DDS_DATAWRITER_QOS_DEFAULT
- DDS_DATAREADER_QOS_DEFAULT
- DDS_TOPIC_QOS_DEFAULT

For example, if you want to set a DataWriter’s QoS back to their DDS-specified default values:

```c
datawriter->set_qos(DDS_DATAWRITER_QOS_DEFAULT);
```

Or if you want to reset the default QosPolicies used by a Publisher to create DataWriters back to their DDS-specified default values:

```c
publisher->set_default_datawriter_qos(DDS_DATAWRITER_QOS_DEFAULT);
```

These defaults cannot be used to initialize a QoS structure for an entity. For example, the following is NOT allowed:

```c
DataWriterQos dataWriterQos = DATAWRITER_QOS_DEFAULT;
// modify QoS...
create_datawriter(dataWriterQos);
```

4.2 QosPolicies

Connext DDS’s behavior is controlled by the Quality of Service (QoS) policies of the data communication Entities (DomainParticipant, Topic, Publisher, Subscriber, DataWriter, and DataReader) used in your applications. This section summarizes each of the QosPolicies that you can set for the various Entities.

The QosPolicy class is the abstract base class for all the QosPolicies. It provides the basic mechanism for an application to specify quality of service parameters. Table 4.2 QosPolicies lists each supported QosPolicy (in alphabetical order), provides a summary, and points to a section in the manual that provides further details.

The detailed description of a QosPolicy that applies to multiple Entities is provided in the first chapter that discusses an Entity whose behavior the QoS affects. Otherwise, the discussion of a QosPolicy can be found in the chapter of the particular Entity to which the policy applies. As you will see in the detailed description sections, all QosPolicies have one or more parameters that are used to configure the policy. The how’s and why’s of tuning the parameters are also discussed in those sections.

As first discussed in 2.7.1 Controlling Behavior with Quality of Service (QoS) Policies on page 23, QosPolicies may interact with each other, and certain values of QosPolicies can be incompatible with the values set for other policies.

The set_qos() operation will fail if you attempt to specify a set of values that would result in an inconsistent set of policies. To indicate a failure, set_qos() will return INCONSISTENT_POLICY. 4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property on page 180 provides further information on QoS compatibility within an Entity as well as across matching Entities, as does the discussion/reference section for each QosPolicy listed in Table 4.2 QosPolicies.
The values of some QosPolicies cannot be changed after the Entity is created or after the Entity is enabled. Others may be changed at any time. The detailed section on each QosPolicy states when each policy can be changed. If you attempt to change a QosPolicy after it becomes immutable (because the associated Entity has been created or enabled, depending on the policy), set_qos() will fail with a return code of IMMUTABLE_POLICY.

**Table 4.2 QosPolicies**

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous-Publisher</td>
<td>Configures the mechanism that sends user data in an external middleware thread. See 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345.</td>
</tr>
<tr>
<td>Availability</td>
<td>This QoS policy is used in the context of two features: For a Collaborative DataWriter, specifies the group of Datawriters expected to collaboratively provide data and the timeouts that control when to allow data to be available that may skip DDS samples. For a Durable Subscription, configures a set of Durable Subscriptions on a DataWriter. See 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370.</td>
</tr>
<tr>
<td>Batch</td>
<td>Specifies and configures the mechanism that allows Connext DDS to collect multiple DDS data samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput. See 7.5.2 BATCH QosPolicy (DDS Extension) on page 374.</td>
</tr>
<tr>
<td>Database</td>
<td>Various settings and resource limits used by Connext DDS to control its internal database. See 9.5.1 DATABASE QosPolicy (DDS Extension) on page 637.</td>
</tr>
<tr>
<td>DataReaderProtocol</td>
<td>This QosPolicy configures the Connext DDS on-the-network protocol, RTPS. See 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570.</td>
</tr>
<tr>
<td>DataReaderResourceLimits</td>
<td>Various settings that configure how DataReaders allocate and use physical memory for internal resources. See 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576.</td>
</tr>
<tr>
<td>DataRepresentation</td>
<td>Specifies which versions of the Extended Common Data Representation (CDR) (version 1 or version 2) and which data compression setting algorithms are offered and requested for your data. See 7.5.3 DATA_REPRESENTATION QosPolicy on page 380.</td>
</tr>
<tr>
<td>DataTag</td>
<td>This QosPolicy can be used to associate a set of tags in the form of (name, value) pairs with a DataReader or DataWriter. The Access Control plugin may use these tags to determine publish and subscribe permissions. See 7.5.4 DATATAG QosPolicy on page 388.</td>
</tr>
<tr>
<td>DataWriterProtocol</td>
<td>This QosPolicy configures the Connext DDS on-the-network protocol, RTPS. See 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.</td>
</tr>
<tr>
<td>DataWriterResourceLimits</td>
<td>Controls how many threads can concurrently block on a write() call of this DataWriter. Also controls the number of batches managed by the DataWriter and the instance replacement kind used by the DataWriter. See 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401.</td>
</tr>
<tr>
<td>Deadline</td>
<td>For a DataReader, specifies the maximum expected elapsed time between arriving DDS data samples. For a DataWriter, specifies a commitment to publish DDS samples with no greater elapsed time between them. See 7.5.7 DEADLINE QosPolicy on page 405.</td>
</tr>
</tbody>
</table>
### Table 4.2 QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>DestinationOrder</td>
<td>Controls how Connext DDS will deal with data sent by multiple DataWriters for the same topic. Can be set to &quot;by reception timestamp&quot; or to &quot;by source timestamp.&quot; See 7.5.8 DESTINATION_ORDER QosPolicy on page 408.</td>
</tr>
<tr>
<td>Discovery</td>
<td>Configures the mechanism used by Connext DDS to automatically discover and connect with new remote applications. See 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640.</td>
</tr>
<tr>
<td>DiscoveryConfig</td>
<td>Controls the amount of delay in discovering Entities in the system and the amount of discovery traffic in the network. See 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644.</td>
</tr>
<tr>
<td>DomainParticipantResource-Limits</td>
<td>Various settings that configure how DomainParticipants allocate and use physical memory for internal resources, including the maximum sizes of various properties. See 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654.</td>
</tr>
<tr>
<td>Durability</td>
<td>Specifies whether or not Connext DDS will store and deliver data that were previously published to new DataReaders. See 7.5.9 DURABILITY QosPolicy on page 411.</td>
</tr>
<tr>
<td>DurabilityService</td>
<td>Various settings to configure the external Persistence Service used by Connext DDS for DataWriters with a Durability QoS setting of Persistent Durability. See 7.5.10 DURABILITY SERVICE QosPolicy on page 416.</td>
</tr>
<tr>
<td>EntityFactory</td>
<td>Controls whether or not child Entities are created in the enabled state. See 7.4.2 ENTITYFACTORY QosPolicy on page 348.</td>
</tr>
<tr>
<td>EntityName</td>
<td>Assigns a name and role_name to an Entity. See 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
<tr>
<td>Event</td>
<td>Configures the DomainParticipant’s internal thread that handles timed events. See 9.5.5 EVENT QosPolicy (DDS Extension) on page 662.</td>
</tr>
<tr>
<td>ExclusiveArea</td>
<td>Configures multi-thread concurrency and deadlock prevention capabilities. See 7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) on page 350.</td>
</tr>
<tr>
<td>GroupData</td>
<td>Along with 5.2.1 TOPIC_DATA QosPolicy on page 224 and 7.5.30 USER_DATA QosPolicy on page 465, this QosPolicy is used to attach a buffer of bytes to Connext DDS’s discovery meta-data. See 7.4.4 GROUP_DATA QosPolicy on page 353.</td>
</tr>
<tr>
<td>History</td>
<td>Specifies how much data must be stored by Connext DDS for the DataWriter or DataReader. This QosPolicy affects the 7.5.21 RELIABILITY QosPolicy on page 447 as well as the 7.5.9 DURABILITY QosPolicy on page 411. See 7.5.12 HISTORY QosPolicy on page 420.</td>
</tr>
<tr>
<td>LatencyBudget</td>
<td>Suggestion to Connext DDS on how much time is allowed to deliver data. See 7.5.13 LATENCYBUDGET QoS Policy on page 425.</td>
</tr>
<tr>
<td>Lifespan</td>
<td>Specifies how long Connext DDS should consider data sent by an user application to be valid. See 7.5.14 LIFESPAN QoS Policy on page 425.</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Specifies and configures the mechanism that allows DataReaders to detect when DataWriters become disconnected or &quot;dead.&quot; See 7.5.15 LIVELINESS QosPolicy on page 427.</td>
</tr>
<tr>
<td>Logging</td>
<td>Configures the properties associated with Connext DDS logging. See 9.4.1 LOGGING QosPolicy (DDS Extension) on page 632.</td>
</tr>
</tbody>
</table>
### Table 4.2 QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiChannel</td>
<td>Configures a DataWriter's ability to send data on different multicast groups (addresses) based on the value of the data. See 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432.</td>
</tr>
<tr>
<td>Ownership</td>
<td>Along with Ownership Strength, specifies if DataReaders for a topic can receive data from multiple DataWriters at the same time. See 7.5.17 OWNERSHIP QosPolicy on page 434.</td>
</tr>
<tr>
<td>OwnershipStrength</td>
<td>Used to arbitrate among multiple DataWriters of the same instance of a Topic when Ownership QoSPolicy is EXCLUSIVE. See 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438.</td>
</tr>
<tr>
<td>Partition</td>
<td>Adds string identifiers that are used for matching DataReaders and DataWriters for the same Topic. See 7.4.5 PARTITION QosPolicy on page 356.</td>
</tr>
<tr>
<td>Presentation</td>
<td>Controls how Connext DDS presents data received by an application to the DataReaders of the data. See 7.4.6 PRESENTATION QosPolicy on page 362.</td>
</tr>
<tr>
<td>Profile</td>
<td>Configures the way that XML documents containing QoS profiles are loaded by RTI. See 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633.</td>
</tr>
<tr>
<td>Property</td>
<td>Stores name/value(string) pairs that can be used to configure certain parameters of Connext DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.</td>
</tr>
<tr>
<td>PublishMode</td>
<td>Specifies how Connext DDS sends application data on the network. By default, data is sent in the user thread that calls the DataWriter's write() operation. However, this QoSPolicy can be used to tell Connext DDS to use its own thread to send the data. See 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444.</td>
</tr>
<tr>
<td>ReaderDataLifeCycle</td>
<td>Controls how a DataReader manages the lifecycle of the data that it has received. See 8.6.3 READER_DATA_LIFECYCLE QosPolicy on page 586.</td>
</tr>
<tr>
<td>ReceiverPool</td>
<td>Configures threads used by Connext DDS to receive and process data from transports (for example, UDP sockets). See 9.5.6 RECEIVER_POOL QosPolicy (DDS Extension) on page 664.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Specifies whether or not Connext DDS will deliver data reliably. See 7.5.21 RELIABILITY QosPolicy on page 447.</td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>Controls the amount of physical memory allocated for Entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.</td>
</tr>
<tr>
<td>Service</td>
<td>Intended for use by RTI infrastructure services. User applications should not modify its value. See 7.5.23 SERVICE QosPolicy (DDS Extension) on page 454.</td>
</tr>
<tr>
<td>SystemResourceLimits</td>
<td>Configures DomainParticipant-independent resources used by Connext DDS. Mainly used to change the maximum number of DomainParticipants that can be created within a single process (address space). See 9.4.3 SYSTEM_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 635.</td>
</tr>
<tr>
<td>TimeBasedFilter</td>
<td>Set by a DataReader to limit the number of new data values received over a period of time. See 8.6.4 TIME_BASED_FILTER QosPolicy on page 588.</td>
</tr>
<tr>
<td>TopicData</td>
<td>Along with Group Data QoSPolicy and User Data QoSPolicy, used to attach a buffer of bytes to Connext DDS's discovery meta-data. See 5.2.1 TOPIC_DATA QosPolicy on page 224.</td>
</tr>
</tbody>
</table>
### 4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property

Some QoS Policies that apply to *Entities* on the sending and receiving sides must have their values set in a compatible manner. This is known as the policy’s ‘requested vs. offered’ (RxO) property. *Entities* on the publishing side ‘offer’ to provide a certain behavior. *Entities* on the subscribing side ‘request’ certain behavior. For Connext DDS to connect the sending entity to the receiving entity, the offered behavior must satisfy the requested behavior.

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopicQueryDispatch</td>
<td>Configures the ability of a DataWriter to publish historical samples in response to a TopicQuery. See 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455.</td>
</tr>
<tr>
<td>TransferMode</td>
<td>Configures the properties of a Zero Copy DataWriter. See 7.5.25 TRANSFER_MODE QosPolicy on page 457.</td>
</tr>
<tr>
<td>TransportBuiltIn</td>
<td>Specifies which built-in transport plugins are used. See 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665.</td>
</tr>
<tr>
<td>TransportMulticast</td>
<td>Specifies the multicast address on which a DataReader wants to receive its data. Can specify a port number as well as a subset of the available transports with which to receive the multicast data. See 8.6.5 TRANSPORT_MULTICAST QosPolicy on page 591.</td>
</tr>
<tr>
<td>TransportMulticastMapping</td>
<td>Specifies the automatic mapping between a list of topic expressions and multicast address that can be used by a DataReader to receive data for a specific topic. See 9.5.8 TRANSPORT_MULTICAST_MAPPING QosPolicy on page 591.</td>
</tr>
<tr>
<td>TransportPriority</td>
<td>Set by a DataReader or DataWriter to tell Connext DDS that the data being sent is a different &quot;priority&quot; than other data. See 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458.</td>
</tr>
<tr>
<td>TransportSelection</td>
<td>Allows you to select which physical transports a DataWriter or DataReader may use to send or receive its data. See 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459.</td>
</tr>
<tr>
<td>TransportUnicast</td>
<td>Specifies a subset of transports and port number that can be used by an Entity to receive data. See 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460.</td>
</tr>
<tr>
<td>TypeConsistencyEnforcement</td>
<td>Defines rules that determine whether the type used to publish a given data stream is consistent with that used to subscribe to it. See 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594.</td>
</tr>
<tr>
<td>TypeSupport</td>
<td>Used to attach application-specific value(s) to a DataWriter or DataReader. These values are passed to the serialization or deserialization routine of the associated data type. Also controls whether padding bytes are set to 0 during serialization. See 7.5.29 TYPESUPPORT QosPolicy (DDS Extension) on page 464.</td>
</tr>
<tr>
<td>UserData</td>
<td>Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to Connext DDS’s discovery meta-data. See 7.5.30 USER_DATA QosPolicy on page 465.</td>
</tr>
<tr>
<td>WireProtocol</td>
<td>Specifies IDs used by the RTPS wire protocol to create globally unique identifiers. See 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670.</td>
</tr>
<tr>
<td>WriterDataLifecycle</td>
<td>Controls how a DataWriter handles the lifecycle of the instances (keys) that the DataWriter is registered to manage. See 7.5.31 WRITER_DATA_LIFECYCLE QosPolicy on page 467.</td>
</tr>
</tbody>
</table>

#### Table 4.2 QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopicQueryDispatch</td>
<td>Configures the ability of a DataWriter to publish historical samples in response to a TopicQuery. See 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455.</td>
</tr>
<tr>
<td>TransferMode</td>
<td>Configures the properties of a Zero Copy DataWriter. See 7.5.25 TRANSFER_MODE QosPolicy on page 457.</td>
</tr>
<tr>
<td>TransportBuiltIn</td>
<td>Specifies which built-in transport plugins are used. See 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665.</td>
</tr>
<tr>
<td>TransportMulticast</td>
<td>Specifies the multicast address on which a DataReader wants to receive its data. Can specify a port number as well as a subset of the available transports with which to receive the multicast data. See 8.6.5 TRANSPORT_MULTICAST QosPolicy on page 591.</td>
</tr>
<tr>
<td>TransportMulticastMapping</td>
<td>Specifies the automatic mapping between a list of topic expressions and multicast address that can be used by a DataReader to receive data for a specific topic. See 9.5.8 TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension) on page 591.</td>
</tr>
<tr>
<td>TransportPriority</td>
<td>Set by a DataReader or DataWriter to tell Connext DDS that the data being sent is a different &quot;priority&quot; than other data. See 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458.</td>
</tr>
<tr>
<td>TransportSelection</td>
<td>Allows you to select which physical transports a DataWriter or DataReader may use to send or receive its data. See 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459.</td>
</tr>
<tr>
<td>TransportUnicast</td>
<td>Specifies a subset of transports and port number that can be used by an Entity to receive data. See 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460.</td>
</tr>
<tr>
<td>TypeConsistencyEnforcement</td>
<td>Defines rules that determine whether the type used to publish a given data stream is consistent with that used to subscribe to it. See 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594.</td>
</tr>
<tr>
<td>TypeSupport</td>
<td>Used to attach application-specific value(s) to a DataWriter or DataReader. These values are passed to the serialization or deserialization routine of the associated data type. Also controls whether padding bytes are set to 0 during serialization. See 7.5.29 TYPESUPPORT QosPolicy (DDS Extension) on page 464.</td>
</tr>
<tr>
<td>UserData</td>
<td>Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to Connext DDS’s discovery meta-data. See 7.5.30 USER_DATA QosPolicy on page 465.</td>
</tr>
<tr>
<td>WireProtocol</td>
<td>Specifies IDs used by the RTPS wire protocol to create globally unique identifiers. See 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670.</td>
</tr>
<tr>
<td>WriterDataLifecycle</td>
<td>Controls how a DataWriter handles the lifecycle of the instances (keys) that the DataWriter is registered to manage. See 7.5.31 WRITER_DATA_LIFECYCLE QosPolicy on page 467.</td>
</tr>
</tbody>
</table>
4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property

For some QosPolicies, the allowed values may be graduated in a way that the offered value will satisfy the requested value if the offered value is either greater than or less than the requested value. For example, if a DataWriter’s DEADLINE QosPolicy specifies a duration less than or equal to a DataReader’s DEADLINE QosPolicy, then the DataWriter is promising to publish data at least as fast or faster than the DataReader requires new data to be received. This is a compatible situation (see 7.5.7 DEADLINE QosPolicy on page 405).

Other QosPolicies require the values on the sending side and the subscribing side to be exactly equal for compatibility to be met. For example, if a DataWriter’s OWNERSHIP QosPolicy is set to SHARED, and the matching DataReader’s value is set to EXCLUSIVE, then this is an incompatible situation since the DataReader and DataWriter have different expectations of what will happen if more than one DataWriter publishes an instance of the Topic (see 7.5.17 OWNERSHIP QosPolicy on page 434).

Finally there are QosPolicies that do not require compatibility between the sending entity and the receiving entity, or that only apply to one side or the other. Whether or not related Entities on the publishing and subscribing sides must use compatible settings for a QosPolicy is indicated in the policy’s RxO property, which is provided in the detailed section on each QosPolicy.

- **RxO = YES** The policy is set at both the publishing and subscribing ends and the values must be set in a compatible manner. What it means to be compatible is defined by the QosPolicy.

- **RxO = NO** The policy is set only on one end or at both the publishing and subscribing ends, but the two settings are independent. There the requested vs. offered semantics are not used for these QosPolicies.

For those QosPolicies that follow the RxO semantics, Connext DDS will compare the values of those policies for compatibility. If they are compatible, then Connext DDS will connect the sending entity to the receiving entity allowing data to be sent between them. If they are found to be incompatible, then Connext DDS will not interconnect the Entities preventing data to be sent between them.

In addition, Connext DDS will record this event by changing the associated communication status in both the sending and receiving applications, see 4.3.1 Types of Communication Status on page 183. Also, if you have installed Listeners on the associated Entities, then Connext DDS will invoke the associated callback functions to notify user code that an incompatible QoS combination has been found, see 4.4.1 Types of Listeners on page 190.

For Publishers and DataWriters, the status corresponding to this situation is **OFFERED_INCOMPATIBLE_QOS_STATUS**. For Subscribers and DataReaders, the corresponding status is **REQUESTED_INCOMPATIBLE_QOS_STATUS**. The question of why a DataReader is not receiving data sent from a matching DataWriter can often be answered if you have instrumented the application with Listeners for the statuses noted previously.
4.2.2 Special QosPolicy Handling Considerations for C

Many QosPolicy structures contain variable-length sequences to store their parameters. In the C++, C++/CLI, C# and Java languages, the memory allocation related to sequences are handled automatically through constructors/destructors and overloaded operators. However, the C language is limited in what it provides to automatically handle memory management. Thus, Connext DDS provides functions and macros in C to initialize, copy, and finalize (free) QosPolicy structures defined for Entities.

In the C language, it is not safe to use an Entity’s QosPolicy structure declared in user code unless it has been initialized first. In addition, user code should always finalize an Entity’s QosPolicy structure to release any memory allocated for the sequences—even if the Entity’s QosPolicy structure was declared as a local, stack variable.

Thus, for a general Entity’s QosPolicy, Connext DDS will provide:

- **DDS_<Entity>Qos_INITIALIZER** This is a macro that should be used when a DDS_<Entity>Qos structure is declared in a C application.

```c
struct DDS_<Entity>Qos qos = DDS_<Entity>Qos_INITIALIZER;
```

- **DDS_<Entity>Qos_initilize()** This is a function that can be used to initialize a DDS_<Entity>Qos structure instead of the macro above.

```c
struct DDS_<Entity>Qos qos;
DDS_<Entity>QOS_initialize(&qos);
```

- **DDS_<Entity>Qos_finalize()** This is a function that should be used to finalize a DDS_<Entity>Qos structure when the structure is no longer needed. It will free any memory allocated for sequences contained in the structure.

```c
struct DDS_<Entity>Qos qos = DDS_<Entity>Qos_INITIALIZER;
...
<use qos>
...
// now done with qos
DDS_<Entity>Qos_finalize(&qos);
```

- **DDS<Entity>Qos_copy()** This is a function that can be used to copy one DDS_<Entity>Qos structure to another. It will copy the sequences contained in the source structure and allocate memory for sequence elements if needed. In the code below, both dstQos and srcQos must have been initialized at some point earlier in the code.

```c
DDS_<Entity>QOS_copy(&dstQos, &srcQos);
```
4.3 Statuses

This section describes the different *statuses* that exist for an entity. A status represents a state or an event regarding the entity. For instance, maybe Connext DDS found a matching DataReader for a DataWriter, or new data has arrived for a DataReader.

Your application can retrieve an *Entity’s* status by:

- explicitly checking for *any* status changes with `get_status_changes()`.
- explicitly checking a *specific* status with `get_<status_name>_status()`.
- using a *Listener*, which provides asynchronous notification when a status changes.
- using *StatusConditions* and *WaitSets*, which provide a way to wait for status changes.

If you want your application to be notified of status changes asynchronously: create and install a *Listener* for the *Entity*. Then internal Connext DDS threads will call the listener methods when the status changes. See 4.4 Listeners on page 189.

If you want your application to wait for status changes: set up *StatusConditions* to indicate the statuses of interest, attach the *StatusConditions* to a *WaitSet*, and then call the *WaitSet’s* `wait()` operation. The call to `wait()` will block until statuses in the attached *Conditions* changes (or until a timeout period expires). See 4.6 Conditions and WaitSets on page 202.

4.3.1 Types of Communication Status

Each *Entity* is associated with a set of *Status* objects representing the “communication status” of that *Entity*. The list of statuses actively monitored by Connext DDS is provided in Table 4.3 Communication Statuses. A status structure contains values that give you more information about the status; for example, how many times the event has occurred since the last time the user checked the status, or how many time the event has occurred in total.

Changes to status values cause activation of corresponding *StatusCondition* objects and trigger invocation of the corresponding *Listener* functions to asynchronously inform the application that the status has changed. For example, a change in a *Topic’s* `INCONSISTENT_TOPIC_STATUS` may trigger the *TopicListener’s* `on_inconsistent_topic()` callback routine (if such a *Listener* is installed).

Table 4.3 Communication Statuses

<table>
<thead>
<tr>
<th>Related Entity</th>
<th>Status (DDS_* STATUS)</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Topic</td>
<td>INCONSISTENT_TOPIC</td>
<td>Another Topic exists with the same name but different characteristics—for example, a different type.</td>
<td>5.3.1 INCONSISTENT_TOPIC Status on page 226</td>
</tr>
</tbody>
</table>
### Table 4.3 Communication Statuses

<table>
<thead>
<tr>
<th>Related Entity</th>
<th>Status (DDS_* _STATUS)</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataWriter</td>
<td>APPLICATION_ACKNOWLEDGMENT</td>
<td>This status indicates that a DataWriter has received an application-level acknowledgment for a DDS sample. The listener provides the identities of the DDS sample and acknowledging DataReader, as well as user-specified response data sent from the DataReader by the acknowledgment message.</td>
<td>7.3.12 Application Acknowledgment on page 317</td>
</tr>
<tr>
<td></td>
<td>DATA_WRITER_CACHE</td>
<td>The status of the DataWriter’s cache. This status does not have a Listener.</td>
<td>7.3.6.2 DATA_WRITER_CACHE_STATUS on page 297</td>
</tr>
<tr>
<td></td>
<td>DATA_WRITER_PROTOCOL</td>
<td>The status of a DataWriter’s internal protocol related metrics (such as the number of DDS samples pushed, pulled, filtered) and the status of wire protocol traffic. This status does not have a Listener.</td>
<td>7.3.6.3 DATA_WRITER_PROTOCOL_STATUS on page 298</td>
</tr>
<tr>
<td></td>
<td>LIVELINESS_LOST</td>
<td>The liveliness that the DataWriter has committed to (through its Liveliness QosPolicy) was not respected (assert_liveliness() or write() not called in time), thus DataReaders may consider the DataWriter as no longer active.</td>
<td>7.3.6.4 LIVELINESS_LOST Status on page 302</td>
</tr>
<tr>
<td></td>
<td>OFFERED_DEADLINE_MISSED</td>
<td>The deadline that the DataWriter has committed through its Deadline QosPolicy was not respected for a specific instance of the Topic.</td>
<td>7.3.6.5 OFFERED_DEADLINE_MISSED Status on page 303</td>
</tr>
<tr>
<td></td>
<td>OFFERED_INCOMPATIBLE_QOS</td>
<td>An offered QosPolicy value was incompatible with what was requested by a DataReader of the same Topic.</td>
<td>7.3.6.6 OFFERED_INCOMPATIBLE_QOS Status on page 303</td>
</tr>
<tr>
<td></td>
<td>PUBLICATION_MATCHED</td>
<td>The DataWriter found a DataReader that matches the Topic, has compatible QoSs and a common partition, or a previously matched DataReader has been deleted.</td>
<td>7.3.6.7 PUBLICATION_MATCHED Status on page 304</td>
</tr>
<tr>
<td></td>
<td>RELIABLE_WRITER_CACHE_CHANGED</td>
<td>The number of unacknowledged DDS samples in a reliable DataWriter’s cache has reached one of the predefined trigger points.</td>
<td>7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305</td>
</tr>
<tr>
<td></td>
<td>RELIABLE_READER_ACTIVITY_CHANGED</td>
<td>One or more reliable DataReaders has either been discovered, deleted, or changed between active and inactive state as specified by the LivelinessQosPolicy of the DataReader.</td>
<td>7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) on page 307</td>
</tr>
<tr>
<td>Subscriber</td>
<td>DATA_ON_READERS</td>
<td>New data is available for any of the readers that were created from the Subscriber.</td>
<td>8.2.9 Statuses for Subscribers on page 503</td>
</tr>
</tbody>
</table>
### Table 4.3 Communication Statuses

<table>
<thead>
<tr>
<th>Related Entity</th>
<th>Status (DDS_*_STATUS)</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataReader</td>
<td>DATA_AVAILABLE</td>
<td>New data (one or more DDS samples) are available for the specific DataReader.</td>
<td>8.3.7.1 DATA_AVAILABLE Status on page 517</td>
</tr>
<tr>
<td></td>
<td>DATA_READER_CACHE</td>
<td>The status of the reader's cache. This status does not have a Listener.</td>
<td>8.3.7.2 DATA_READER_CACHE_STATUS on page 517</td>
</tr>
<tr>
<td></td>
<td>DATA_READER_PROTOCOL</td>
<td>The status of a DataReader's internal protocol related metrics (such as the number of DDS samples received, filtered, rejected) and the status of wire protocol traffic. This status does not have a Listener.</td>
<td>8.3.7.3 DATA_READER_PROTOCOL_STATUS on page 520</td>
</tr>
<tr>
<td></td>
<td>LIVELINESS_CHANGED</td>
<td>The liveliness of one or more DataWriters that were writing instances read by the DataReader has either been discovered, deleted, or changed between active and inactive state as specified by the LivelinessQosPolicy of the DataWriter.</td>
<td>8.3.7.4 LIVELINESS_CHANGED Status on page 524</td>
</tr>
<tr>
<td>DataReader cont'd</td>
<td>REQUESTED_DEADLINE_MISSED</td>
<td>New data was not received for an instance of the Topic within the time period set by the DataReader's Deadline QosPolicy.</td>
<td>8.3.7.5 REQUESTED_DEADLINE_MISSED Status on page 526</td>
</tr>
<tr>
<td></td>
<td>REQUESTED_INCOMPATIBLE_QOS</td>
<td>A requested QosPolicy value was incompatible with what was offered by a DataWriter of the same Topic.</td>
<td>8.3.7.6 REQUESTED_INCOMPATIBLE_QOS Status on page 526</td>
</tr>
<tr>
<td></td>
<td>SAMPLE_LOST</td>
<td>A DDS sample sent by Connext DDS has been lost (never received).</td>
<td>8.3.7.7 SAMPLE_LOST Status on page 527</td>
</tr>
<tr>
<td></td>
<td>SAMPLE_REJECTED</td>
<td>A received DDS sample has been rejected due to a resource limit (buffers filled).</td>
<td>8.3.7.8 SAMPLE_REJECTED Status on page 531</td>
</tr>
<tr>
<td></td>
<td>SUBSCRIPTION_MATCHED</td>
<td>The DataReader has found a DataWriter that matches the Topic, has compatible QoSs and a common partition, or an existing matched DataWriter has been deleted.</td>
<td>8.3.7.9 SUBSCRIPTION_MATCHED Status on page 533</td>
</tr>
</tbody>
</table>

Statuses can be grouped into two categories:

- **Plain communication status:**

  In addition to a flag that indicates whether or not a status has changed, a plain communication status also contains state and thus has a corresponding structure to hold its current value.
4.3.1 Types of Communication Status

- Read communication status:

  A read communication status is more like an event and has no state other than whether or not it has occurred. Only two statuses listed in Table 4.3 Communication Statuses are read communications statuses: DATA_AVAILABLE and DATA_ON_READERS.

As mentioned in 4.1.4 Getting Status and Status Changes on page 171, all Entities have a get_status_changes() operation that can be used to explicitly poll for changes in any status related to the entity. For plain statuses, each entry has operations to get the current value of the status; for example, the Topic class has a get_inconsistent_topic_status() operation. For read statuses, your application should use the take() operation on the DataReader to retrieve the newly arrived data that is indicated by DATA_AVAILABLE and DATA_ON_READERS.

Note that the two read communication statuses do not change independently. If data arrives for a DataReader, then its DATA_AVAILABLE status changes. At the same time, the DATA_ON_READERS status changes for the DataReader’s Subscriber.

Both types of status have a StatusChangedFlag. This flag indicates whether that particular communication status has changed since the last time the status was read by the application. The way the StatusChangedFlag is maintained is slightly different for the plain communication status and the read communication status, as described in the following sections:

- 4.3.1.1 Changes in Plain Communication Status below
- 4.3.1.2 Changes in Read Communication Status on the next page

4.3.1.1 Changes in Plain Communication Status

As seen in Figure 4.2: Status Changes for Plain Communication Status below, for the plain communication status, the StatusChangedFlag flag is initially set to FALSE. It becomes TRUE whenever the plain communication status changes and is reset to FALSE each time the application accesses the plain communication status via the proper get_*_status() operation.

Figure 4.2: Status Changes for Plain Communication Status

![](image)

The communication status is also reset to FALSE whenever the associated listener operation is called, as the listener implicitly accesses the status which is passed as a parameter to the operation.
The fact that the status is reset prior to calling the listener means that if the application calls the `get_*_status` operation from inside the listener, it will see the status already reset.

An exception to this rule is when the associated listener is the 'nil' listener. The 'nil' listener is treated as a NO-OP and the act of calling the 'nil' listener does not reset the communication status. (See 4.4.1 Types of Listeners on page 190.)

For example, the value of the StatusChangedFlag associated with the REQUESTED_DEADLINE_MISSED status will become TRUE each time new deadline occurs (which increases the RequestedDeadlineMissed status’ total_count field). The value changes to FALSE when the application accesses the status via the corresponding `get_requested_deadline_missed_status` operation on the proper Entity.

### 4.3.1.2 Changes in Read Communication Status

As seen in Figure 4.3: Status Changes for Read Communication Status on the next page, for the read communication status, the StatusChangedFlag flag is initially set to FALSE. The StatusChangedFlag becomes TRUE when either a DDS data sample arrives or the ViewStateKind, SampleStateKind, or InstanceStateKind of any existing DDS sample changes for any reason other than a call to one of the read/take operations. Specifically, any of the following events will cause the StatusChangedFlag to become TRUE:

- The arrival of new data.
- A change in the InstanceStateKind of a contained instance. This can be caused by either:
  - Notification that an instance has been disposed by:
    - the DataWriter that owns it, if OWNERSHIP = EXCLUSIVE
    - or by any DataWriter, if OWNERSHIP = SHARED
  - The loss of liveliness of the DataWriter of an instance for which there is no other DataWriter.
  - The arrival of the notification that an instance has been unregistered by the only DataWriter that is known to be writing the instance.

Depending on the kind of StatusChangedFlag, the flag transitions to FALSE (that is, the status is reset) as follows:

- The DATA_AVAILABLE StatusChangedFlag becomes FALSE when either on_data_available() is called or the read/take operation (or their variants) is called on the associated DataReader.
- The DATA_ON_READERS StatusChangedFlag becomes FALSE when any of the following occurs:
  - on_data_on_readers() is called.
  - on_data_available() is called on any DataReader belonging to the Subscriber.
- **read(), take(),** or one of their variants is called on any `DataReader` belonging to the `Subscriber`.

**Figure 4.3: Status Changes for Read Communication Status**

4.3.2 Special Status-Handling Considerations for C

Some status structures contain variable-length sequences to store their values. In the C++, C++/CLI, C# and Java languages, the memory allocation related to sequences are handled automatically through constructors/destructors and overloaded operators. However, the C language is limited in what it provides to
automatically handle memory management. Thus, Connext DDS provides functions and macros in C to initialize, copy, and finalize (free) status structures.

In the C language, it is not safe to use a status structure that has internal sequences declared in user code unless it has been initialized first. In addition, user code should always finalize a status structure to release any memory allocated for the sequences—even if the status structure was declared as a local, stack variable.

Thus, for a general status structure, Connext DDS will provide:

- **DDS_<STATUS>_STATUS_INITIALIZER** This is a macro that should be used when a DDS_<Status> structure is declared in a C application.
  
  ```
  struct DDS_<Status> Status status = DDS_<Status>_STATUS_INITIALIZER;
  ```

- **DDS_<Status>_Status_initialize()** This is a function that can be used to initialize a DDS_<Status> structure instead of the macro above.
  
  ```
  struct DDS_<Status> Status status;
  DDS_<Status>_Status_initialize(&status);
  ```

- **DDS_<Status>_Status_finalize()** This is a function that should be used to finalize a DDS_<Status> structure when the structure is no longer needed. It will free any memory allocated for sequences contained in the structure.
  
  ```
  struct DDS_<Status> Status status = DDS_<Status>_STATUS_INITIALIZER;
  ...
  <use status>
  ...
  // now done with Status
  DDS_<Status>_Status_finalize(&status);
  ```

- **DDS<Status>_Status_copy()** This is a function that can be used to copy one DDS_<Status> structure to another. It will copy the sequences contained in the source structure and allocate memory for sequence elements if needed. In the code below, both dstStatus and srcStatus must have been initialized at some point earlier in the code.
  
  ```
  DDS_<Status>_Status_copy(&dstStatus, &srcStatus);
  ```

Note that many status structures do not have sequences internally. For those structures, you do not need to use the macro and methods provided above. However, they have still been created for your convenience.

### 4.4 Listeners

Listeners are triggered by changes in an entity’s status. For instance, maybe Connext DDS found a matching DataReader for a DataWriter, or new data has arrived for a DataReader.

You can use either Listeners or WaitSets to be notified of events. WaitSets block a thread until data is available. This is the safest way to get data, because it does not affect any middleware threads. In contrast, Listeners allow an application to be called back from a Connext DDS thread. This provides better latency
than *WaitSets*, because the application can handle the event in the same thread that is generating the notification (so there is no time spent context-switching between threads).

There is also the possibility that notifications can be lost when using *WaitSets*, because most notifications contain a status update for only the most recent event. For example, imagine a system where a *DataReader* is trying to detect that *DataWriters* have lost liveliness. If two *DataWriters* lose liveliness at about the same, a listener that handles the **on_liveliness_changed** status will be called back once for each *DataWriter* that lost liveliness. When **on_liveliness_changed** is called back the first time, the LivelinessChangedStatus will contain the handle for one of the *DataWriters*, and the second time the callback is called it will contain the handle for the other *DataWriter*. However, if *WaitSets* are used and the *DataWriters* become not alive at about the same time, it's possible that by the time the *WaitSet* is notified that the first *DataWriter* has lost liveliness, the second one also loses liveliness, and the LivelinessChangedStatus contains only the most recent *DataWriter* to lose liveliness.

The danger of using *Listeners* is that they are called back from a *Connext DDS* thread, so performing any slow processing in a *Listener* callback can degrade the performance of *Connext DDS* (by causing lost data, lost liveliness, etc.).

This section describes *Listeners* and how to use them.

**4.4.1 Types of Listeners**

The *Listener* class is the abstract base class for all listeners. Each entity class (*DomainParticipant, Topic, Publisher, DataWriter, Subscriber, and DataReader*) has its own derived *Listener* class that add methods for handling entity-specific statuses. The hierarchy of *Listener* classes is presented in Figure 4.4: Listener Class Hierarchy on the next page. The methods are called by an internal *Connext DDS* thread when the corresponding status for the *Entity* changes value.
4.4.1 Types of Listeners

Figure 4.4: Listener Class Hierarchy

You can choose which changes in status will trigger a callback by installing a listener with a bit-mask. Bits in the mask correspond to different statuses. The bits that are true indicate that the listener will be called back when there are changes in the corresponding status.

You can specify a listener and set its bit-mask before or after you create an Entity:

**During Entity creation:**

```java
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS | DDS_DATA_AVAILABLE_STATUS;
datreader = subscriber->create_datareader(topic,
          DDS_DATAREADER_QOS_DEFAULT,
          listener, mask);
```
4.4.2 Creating and Deleting Listeners

or afterwards:

```c
dds_statusmask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
                    DDS_DATA_AVAILABLE_STATUS;

DataReader->set_listener(listener, mask);
```

As you can see in the above examples, there are two components involved when setting up listeners: the listener itself and the mask. Both of these can be null. Table 4.4 Effect of Different Combinations of Listeners and Status Bit Masks describes what happens when a status change occurs. See 4.4.5 Hierarchical Processing of Listeners on page 194 for more information.

<table>
<thead>
<tr>
<th>Listener is Specified</th>
<th>No Bits Set in Mask</th>
<th>Some/All Bits Set in Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Listener is Specified</strong></td>
<td><strong>Connext DDS finds the next most relevant listener for the changed status.</strong></td>
<td>For the statuses that are enabled in the mask, the most relevant listener will be called. The 'statusChangedFlag' for the relevant status is reset.</td>
</tr>
<tr>
<td><strong>Listener is NULL</strong></td>
<td><strong>Connext DDS behaves as if the listener is not installed and finds the next most relevant listener for that status.</strong></td>
<td><strong>Connext DDS behaves as if the listener callback is installed, but the callback is doing nothing. This is called a 'nil' listener.</strong></td>
</tr>
</tbody>
</table>

### 4.4.2 Creating and Deleting Listeners

There is no factory for creating or deleting a Listener; use the natural means in each language binding (for example, “new” or “delete” in C++ or Java). For example:

```c
class HelloWorldListener : public DDSDataReaderListener {
    virtual void on_data_available(DDSDataReader* reader);
};

void HelloWorldListener::on_data_available(DDSDataReader* reader)
{
    printf("received data\n");
}

// Create a Listener
HelloWorldListener *reader_listener = NULL;
reader_listener = new HelloWorldListener();
// Delete a Listener
delete reader_listener;
```

A listener cannot be deleted until the entity it is attached to has been deleted. For example, you must delete the DataReader before deleting the DataReader’s listener.

**Note:** Due to a thread-safety issue, the destruction of a DomainParticipantListener from an enabled DomainParticipant should be avoided—even if the DomainParticipantListener has been removed from the DomainParticipant. (This limitation does not affect the Java API.)
4.4.3 Special Considerations for Listeners in C

In C, a Listener is a structure with function pointers to the user callback routines. Often, you may only be interested in a subset of the statuses that can be monitored with the Listener. In those cases, you may not set all of the functions pointers in a listener structure to a valid function. In that situation, we recommend that the unused, callback-function pointers are set to NULL. While setting the DDS_StatusMask to enable only the callbacks for the statuses in which you are interested (and thus only enabling callbacks on the functions that actually exist) is safe, we still recommend that you clear all of the unused callback pointers in the Listener structure.

To help, in the C language, we provide a macro that can be used to initialize a Listener structure so that all of its callback pointers are set to NULL. For example

```c
DDS_<Entity>Listener listener = DDS_<Entity>Listener_INITIALIZER;
// now only need to set the listener callback pointers for statuses // to be monitored
```

There is no need to do this in languages other than C.

4.4.4 Special Considerations for Listeners in Modern C++

In the Modern C++ API, the Entity constructors and set_listener functions expect a std::shared_ptr. The Entity keeps a reference to that shared_ptr preventing its deletion at least until the Entity has been destroyed or the Listener reset.

In addition to each Listener base class (such as dds::sub::DataReaderListener), which contains a set of pure virtual functions, a class with a default implementation for each callback that does nothing is provided (such as dds::sub::NoOpDataReaderListener).

The following example shows how to create a DataReaderListener:

```c++
class HelloWorldListener : public dds::sub::NoOpDataReaderListener<HelloWorld> {
  void on_data_available(dds::sub::DataReader<HelloWorld> reader) override {
    auto samples = reader.take();
    std::cout << "Received " << samples.length() << " samples\n";
  }
};
void create_reader_with_listener() {
  // ...
  auto reader_listener = std::make_shared<HelloWorldListener>();
  dds::sub::DataReader<HelloWorld> reader(subscriber, topic, qos, reader_listener);
  // ...
}
```

It is not recommended to keep a reference to the Entity as a member of a Listener class. Doing so creates a cycle between these two references preventing each other's destruction. If you do need to keep a reference, you must later reset the Listener or explicitly close the Entity.
4.4.5 Hierarchical Processing of Listeners

As seen in Figure 4.4: Listener Class Hierarchy on page 191, Listeners for some Entities derive from the Connext DDS Listeners for related Entities. This means that the derived Listener has all of the methods of its parent class. You can install Listeners at all levels of the object hierarchy. At the top is the DomainParticipantListener; only one can be installed in a DomainParticipant. Then every Subscriber and Publisher can have their own Listener. Finally, each Topic, DataReader and DataWriter can have their own listeners. All are optional.

Suppose, however, that an Entity does not install a Listener, or installs a Listener that does not have particular communication status selected in the bitmask. In this case, if/when that particular status changes for that Entity, the corresponding Listener for that Entity’s parent is called. Status changes are “propagated” from child Entity to parent Entity until a Listener is found that is registered for that status. Connext DDS will give up and drop the status-change event only if no Listeners have been installed in the object hierarchy to be called back for the specific status. This is true for plain communication statuses. Read communication statuses are handle somewhat differently, see 4.4.5.1 Processing Read Communication Statuses on the next page.

For example, suppose that Connext DDS finds a matching DataWriter for a local DataReader. This event will change the SUBSCRIPTION_MATCHED status. So the local DataReader object is checked to see if the application has installed a listener that handles the SUBSCRIPTION_MATCH status. If not, the Subscriber that created the DataReader is checked to see if it has a listener installed that handles the same event. If not, the DomainParticipant is checked. The DomainParticipantListener methods are called only if none of the descendent Entities of the DomainParticipant have listeners that handle the particular status that has changed. Again, all listeners are optional. Your application does not have to handle any communication statuses.

Table 4.5 Listener Callback Functions lists the callback functions that are available for each Entity’s status listener.
4.4.5 Hierarchical Processing of Listeners

Table 4.5 Listener Callback Functions

<table>
<thead>
<tr>
<th>Entity Listener for:</th>
<th>Callback Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Topics</td>
</tr>
<tr>
<td></td>
<td>on_inconsistent_topic()</td>
</tr>
<tr>
<td></td>
<td>on_liveliness_lost()</td>
</tr>
<tr>
<td></td>
<td>on_offered_deadline_missed()</td>
</tr>
<tr>
<td></td>
<td>on_offered_incompatible_qos()</td>
</tr>
<tr>
<td></td>
<td>on_publication_matched()</td>
</tr>
<tr>
<td></td>
<td>on_reliable_reader_activity_changed()</td>
</tr>
<tr>
<td></td>
<td>on_reliable_writer_cache_changed()</td>
</tr>
<tr>
<td>Publishers and DataWriters</td>
<td>on_data_on_readers()</td>
</tr>
<tr>
<td>Subscribers and DataReaders</td>
<td>on_data_available</td>
</tr>
<tr>
<td></td>
<td>on_liveliness_changed()</td>
</tr>
<tr>
<td></td>
<td>on_requested_deadline_missed()</td>
</tr>
<tr>
<td></td>
<td>on_requested_incompatible_qos()</td>
</tr>
<tr>
<td></td>
<td>on_sample_lost()</td>
</tr>
<tr>
<td></td>
<td>on_sample_rejected()</td>
</tr>
<tr>
<td></td>
<td>on_subscription_matched()</td>
</tr>
</tbody>
</table>

4.4.5.1 Processing Read Communication Statuses

The processing of the DATA_ON_READERS and DATA_AVAILABLE read communication statuses are handled slightly differently since, when new data arrives for a DataReader, both statuses change simultaneously. However, only one, if any, Listener will be called to handle the event.

If there is a Listener installed to handle the DATA_ON_READERS status in the DataReader’s Subscriber or in the DomainParticipant, then that Listener’s on_data_on_readers() function will be called back. The DataReaderListener’s on_data_available() function is called only if the DATA_ON_READERS status is not handle by any relevant listeners.

This can be useful if you have generic processing to do whenever new data arrives for any DataReader. You can execute the generic code in the on_data_on_readers() method, and then dispatch the processing of the actual data to the specific DataReaderListener’s on_data_available() function by calling the notify_datareaders() method on the Subscriber.

For example:
void on_data_on_readers (DDSSubscriber *subscriber)
{
    // Do some general processing that needs to be done
    // whenever new data arrives, but is independent of
    // any particular DataReader
    < generic processing code here >

    // Now dispatch the actual processing of the data
    // to the specific DataReader for which the data
    // was received
    subscriber->notify_datareaders();
}

4.4.6 Operations Allowed within Listener Callbacks

Due to the potential for deadlock, some Connext DDS APIs should not be invoked within the functions of listener callbacks. Exactly which Connext DDS APIs are restricted depends on the Entity upon which the Listener is installed, as well as the configuration of ‘Exclusive Areas,’ as discussed in 4.5 Exclusive Areas (EAs) on the next page.

Please read and understand 4.5 Exclusive Areas (EAs) on the next page and 4.5.1 Restricted Operations in Listener Callbacks on page 200 to ensure that the calls made from your Listeners are allowed and will not cause potential deadlock situations.

4.4.7 Best Practices with Listeners

Note: All the issues described below can be avoided by using a Waitset.

- Avoid blocking or performing a lot of processing in Listener callbacks

Listeners are invoked by internal threads that perform critical functions within the middleware and need to run in a timely manner (see Connext DDS Threading Model (Chapter 21 on page 908)). By default, Connext DDS creates a few threads to use to receive data and only a single thread to handle periodic events.

Because of this, user applications installing Listeners should never block in a Listener callback. There are several negative consequences of blocking in a listener callback:

- The application may lose data for the DataReader the listener is installed on, because the receive thread is not removing it from the socket buffer and it gets overwritten (see 21.3 Receive Threads on page 911).

- The application may receive strictly reliable data with a delay, because the receive thread is not removing it from the socket buffer and if it gets overwritten it must be re-sent.

- The application may lose or delay data for other DataReaders, because by default all DataReaders created with the same DomainParticipant share the same threads.
- The application may not be notified of periodic events on time (see 21.2 Event Thread on page 910).

  If the application needs to make a blocking call when data is available, or when another event occurs, the application should use a *WaitSet*. (see 4.6 Conditions and WaitSets on page 202).

- Avoid taking application mutexes/semaphores in *Listener* callbacks

  Taking application mutexes/semaphores within a *Listener* callback may lead to unexpected deadlock scenarios. When a *Listener* callback is invoked, the EA (Exclusive Area) of the Entity 'E' to which the callback applies is taken by the middleware. If the application takes an application mutex 'M' within a critical section in which the application makes DDS calls affecting 'E', this may lead to following deadlock:

  The middleware thread is within the entity EA trying to acquire the mutex 'M'. At the same time, the application thread has acquired 'M' and is blocked trying to acquire the entity EA.

- Do not write data with a *DataWriter* within the *on_data_available()* callback

  Avoid writing data with a *DataWriter* within the DataReaderListener's *on_data_available()* callback. If the write operation blocks because e.g. the send window is full, this will lead to a deadlock.

- Do not call *wait_for_acknowledgements()* within the *on_data_available()* callback

  Do not call the *DataWriter*’s *wait_for_acknowledgements()* within the DataReaderListener's *on_data_available()* callback. This will lead to deadlock.

4.5 Exclusive Areas (EAs)

Listener callbacks are invoked by internal *Connext DDS* threads. To prevent undesirable, multi-threaded interaction, the internal threads may take and hold semaphores (mutexes) used for mutual exclusion. In your listener callbacks, you may want to invoke functions provided by the *Connext DDS* API. Internally, those *Connext DDS* functions also may take mutexes to prevent errors due to multi-threaded access to critical data or operations.

Once there are multiple mutexes to protect different critical regions, the possibility for deadlock exists. Consider Figure 4.5: Multiple Mutexes Leading to a Deadlock Condition on the next page’s scenario, in which there are two threads and two mutexes.
**Figure 4.5: Multiple Mutexes Leading to a Deadlock Condition**

Thread1

- Take (MutexA)
- Take (MutexB)

Thread2

- Take (MutexB)
- Take (MutexA)

Deadlock!

*Thread1 takes MutexA while simultaneously Thread2 takes MutexB. Then, Thread1 takes MutexB and simultaneously Thread2 takes MutexA. Now both threads are blocked since they hold a mutex that the other thread is trying to take. This is a deadlock condition.*

While the probability of entering the deadlock situation in Figure 4.5: Multiple Mutexes Leading to a Deadlock Condition above depends on execution timing, when there are multiple threads and multiple mutexes, care must be taken in writing code to prevent those situations from existing in the first place. *Connext DDS* has been carefully created and analyzed so that we know our threads internally are safe from deadlock interactions.

However, when *Connext DDS* threads that are holding mutexes call user code in listeners, it is possible for user code to inadvertently cause the threads to deadlock if *Connext DDS* APIs that try to take other mutexes are invoked. To help you avoid this situation, RTI has defined a concept known as *Exclusive Areas*, some restrictions regarding the use of *Connext DDS* APIs within user callback code, and a QoS policy that allows you to configure *Exclusive Areas*.

*Connext DDS* uses *Exclusive Areas* (EAs) to encapsulate mutexes and critical regions. Only one thread at a time can be executing code within an EA. The formal definition of EAs and their implementation ensures safety from deadlock and efficient entering and exiting of EAs. While every *Entity* created by *Connext DDS* has an associated EA, EAs may be shared among several *Entities*. A thread is automatically in the entity’s EA when it is calling the entity’s listener.

*Connext DDS* allows you to configure all the *Entities* within an application in a single DDS domain to share a single *Exclusive Area*. This would greatly restrict the concurrency of thread execution within *Connext DDS*’s multi-threaded core. However, doing so would release all restrictions on using *Connext DDS* APIs within your callback code.
You may also have the best of both worlds by configuring a set of Entities to share a global EA and others to have their own. For the Entities that have their own EAs, the types of Connext DDS operations that you can call from the Entity’s callback are restricted.

To understand why the general EA framework limits the operations that can be called in an EA, consider a modification to the example previously presented in Figure 4.5: Multiple Mutexes Leading to a Deadlock Condition on the previous page. Suppose we create a rule that is followed when we write our code. “For all situations in which a thread has to take multiple mutexes, we write our code so that the mutexes are always taken in the same order.” Following the rule will ensure us that the code we write cannot enter a deadlock situation due to the taking of the mutexes, see Figure 4.6: Taking Multiple Mutexes in a Specific Order to Eliminate Deadlock below.

Figure 4.6: Taking Multiple Mutexes in a Specific Order to Eliminate Deadlock

<table>
<thead>
<tr>
<th>Thread1</th>
<th>Thread2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Take (MutexA)</td>
<td>Take (MutexA)</td>
</tr>
<tr>
<td>Take (MutexB)</td>
<td></td>
</tr>
<tr>
<td>Give (MutexB)</td>
<td></td>
</tr>
<tr>
<td>Give (MutexA)</td>
<td>Take (MutexB)</td>
</tr>
</tbody>
</table>

By creating an order in which multiple mutexes are taken, you can guarantee that no deadlock situation will arise. In this case, if a thread must take both MutexA and MutexB, we write our code so that in those cases MutexA is always taken before MutexB.

Connext DDS defines an ordering of the mutexes it creates. Generally speaking, there are three ordered levels of Exclusive Areas:

- **ParticipantEA**

  There is only one ParticipantEA per participant. The creation and deletion of all Entities (create_xxx(), delete_xxx()) take the ParticipantEA. In addition, the enable() method for an Entity and the setting of the Entity’s QoS, set_qos(), also take the ParticipantEA. There are other functions that take the ParticipantEA: get_discovered_participants(), get_publishers(), get_subscribers(), get_discovered_topics(), ignore_participant(), ignore_topic(), ignore_publication(), ignore_subscription(), remove_peer(), and register_type().
4.5.1 Restricted Operations in Listener Callbacks

- **SubscriberEA**

  This EA is created on a per-Subscriber basis by default. You can assume that the methods of a Subscriber will take the SubscriberEA. In addition, the DataReaders created by a Subscriber share the EA of its parent. This means that the methods of a DataReader (including `take()` and `read()`) will take the EA of its Subscriber. Therefore, **operations on DataReaders of the same Subscriber, will be serialized**, even when invoked from multiple concurrent application threads. As mentioned, the `enable()` and `set_qos()` methods of both Subscribers and DataReaders will take the ParticipantEA. The same is true for the `create_datareader()` and `delete_datareader()` methods of the Subscriber.

- **PublisherEA**

  This EA is created on a per-Publisher basis by default. You can assume that the methods of a Publisher will take the PublisherEA. In addition, the DataWriters created by a Publisher share the EA of its parent. This means that the methods of a DataWriter including `write()` will take the EA of its Publisher. Therefore, **operations on DataWriters of the same Publisher will be serialized**, even when invoked from multiple concurrent application threads. As mentioned, the `enable()` and `set_qos()` methods of both Publishers and DataWriters will take the ParticipantEA, as well as the `create_datawriter()` and `delete_datawriter()` methods of the Publisher.

In addition, you should also be aware that:

- The three EA levels are ordered in the following manner:
  ParticipantEA < SubscriberEA < PublisherEA

- When executing user code in a listener callback of an Entity, the internal Connext DDS thread is already in the EA of that Entity or used by that Entity.

- If a thread is in an EA, it can call methods associated with either a higher EA level or that share the same EA. It cannot call methods associated with a lower EA level nor ones that use a different EA at the same level.

**4.5.1 Restricted Operations in Listener Callbacks**

Based on the background and rules provided in 4.5 Exclusive Areas (EAs) on page 197, this section describes how EAs restrict you from using various Connext DDS APIs from within the Listener callbacks of different Entities. Reader callbacks take the SubscriberEA. Writer callbacks take the PublisherEA. DomainParticipant callbacks take the ParticipantEA.

These restrictions do not apply to builtin topic listener callbacks.

By default, each Publisher and Subscriber creates and uses its own EA, and shares it with its children DataWriters and DataReaders, respectively. In that case:

Within a DataWriter/DataReader’s Listener callback, do not:
4.5.1 Restricted Operations in Listener Callbacks

- Create any *Entities*
- Delete any *Entities*
- Enable any *Entities*
- Set QoS on any *Entities*

Within a *Subscriber/DataReader’s Listener* callback, do not call any operations on:

- Other *Subscribers*
- *DataReaders* that belong to other *Subscribers*
- *Publishers/DataWriters* that have been configured to use the ParticipantEA (see below)

Within a *Publisher/DataWriter Listener* callback, do not call any operations on:

- Other *Publishers*
- *DataWriters* that belong to other *Publishers*
- Any *Subscribers*
- Any *DataReaders*

*Connext DDS* will enforce the rules to avoid deadlock, and any attempt to call an illegal method from within a *Listener* callback will return DDS_RETCODE_ILLEGAL_OPERATION.

However, as previously mentioned, if you are willing to trade-off concurrency for flexibility, you may configure individual *Publishers* and *Subscribers* (and thus their *DataWriters* and *DataReaders*) to share the EA of their participant. In the limit, only a single ParticipantEA is shared among all *Entities*. When doing so, the restrictions above are lifted at a cost of greatly reduced concurrency. You may create/delete/enable/set_qos’s and generally call all of the methods of any other entity in the Listener callbacks of *Entities* that share the ParticipantEA.

Use the 7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) on page 350 of the *Publisher* or *Subscriber* to set whether or not to use a shared exclusive area. By default, *Publishers* and *Subscribers* will create and use their own individual EAs. You can configure a subset of the *Publishers* and *Subscribers* to share the ParticipantEA if you need the Listeners associated with those *Entities* or child *Entities* to be able to call any of the restricted methods listed above.

Regardless of how the EXCLUSIVE_AREA QosPolicy is set, the following operations are never allowed in any *Listener* callback:

- Destruction of the entity to which the *Listener* is attached. For instance, a *DataWriter/DataReader* Listener callback must not destroy its *DataWriter/DataReader*. 
Within the TopicListener callback, you cannot call any operations on DataReaders, DataWriters, Publishers, Subscribers or DomainParticipants.

4.6 Conditions and WaitSets

Conditions and WaitSets provide another way for Connext DDS to communicate status changes (including the arrival of data) to your application. While a Listener is used to provide a callback for asynchronous access, Conditions and WaitSets provide synchronous data access. In other words, Listeners are notification-based and Conditions are wait-based.

A WaitSet allows an application to wait until one or more attached Conditions becomes true (or until a timeout expires).

Briefly, your application can create a WaitSet, attach one or more Conditions to it, then call the WaitSet's wait() operation. The wait() blocks until one or more of the WaitSet's attached Conditions becomes TRUE.

A Condition has a trigger_value that can be TRUE or FALSE. You can retrieve the current value by calling the Condition’s only operation, get_trigger_value().

There are three kinds of Conditions. A Condition is a root class for all the conditions that may be attached to a WaitSet. This basic class is specialized in three classes:

- **4.6.6 GuardConditions on page 209** are created by your application. Each GuardCondition has a single, user-settable, boolean trigger_value. Your application can manually trigger the GuardCondition by calling set_trigger_value(). Connext DDS does not trigger or clear this type of condition—it is completely controlled by your application.

- **4.6.7 ReadConditions and QueryConditions on page 209** are created by your application, but triggered by Connext DDS. ReadConditions provide a way for you to specify the DDS data samples that you want to wait for, by indicating the desired sample-states, view-states, and instance-states¹.

- **4.6.8 StatusConditions on page 212** are created automatically by Connext DDS, one for each Entity. A StatusCondition is triggered by Connext DDS when there is a change to any of that Entity’s enabled statuses.

Figure 4.7: Conditions and WaitSets on the next page shows the relationship between these objects and other Entities in the system.

¹These states are described in 8.4.6 The SampleInfo Structure on page 565.
A WaitSet can be associated with more than one Entity (including multiple DomainParticipants). It can be used to wait on Conditions associated with different DomainParticipants. A WaitSet can only be in use by one application thread at a time.

### 4.6.1 Creating and Deleting WaitSets

There is no factory for creating or deleting a WaitSet; use the natural means in each language binding (for example, “new” or “delete” in C++ or Java).

There are two ways to create a WaitSet—with or without specifying WaitSet properties (DDS_WaitSetProperty_t, described in Table 4.6 WaitSet Properties (DDS_WaitSet_Property_t)). 4.6.3 Waiting for Conditions on page 205 describes how the properties are used.
4.6.2 WaitSet Operations

### Table 4.6 WaitSet Properties (DDS_WaitSet_Property_t)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>max_event_count</td>
<td>Maximum number of trigger events to cause a WaitSet to wake up.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>max_event_delay</td>
<td>Maximum delay from occurrence of first trigger event to cause a WaitSet to wake up. This value should reflect the maximum acceptable latency increase (time delay from occurrence of the event to waking up the WaitSet) incurred as a result of waiting for additional events before waking up the WaitSet.</td>
</tr>
</tbody>
</table>

To create a WaitSet with default behavior:

```c
WaitSet* waitset = new WaitSet();
```

To create a WaitSet with properties:

```c
DDS_WaitSetProperty_t prop;
Prop.max_event_count = 5;
DDSWaitSet* waitset = new DDSWaitSet(prop);
```

To delete a WaitSet:

```c
delete waitset;
```

### 4.6.2 WaitSet Operations

WaitSets have only a few operations, as listed in Table 4.7 WaitSet Operations. For details, see the API Reference HTML documentation.

### Table 4.7 WaitSet Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>attach_condition</td>
<td>Attaches a Condition to this WaitSet. You may attach a Condition to a WaitSet that is currently being waited upon (via the wait() operation). In this case, if the Condition has a trigger_value of TRUE, then attaching the Condition will unblock the WaitSet. Adding a Condition that is already attached to the WaitSet has no effect. If the Condition cannot be attached, Connext DDS will return an OUT_OF_RESOURCES error code.</td>
</tr>
<tr>
<td>detach_condition</td>
<td>Detaches a Condition from the WaitSet. Attempting to detach a Condition that is not to attached the WaitSet will result in a PRECONDITION_NOT_MET error code.</td>
</tr>
<tr>
<td>wait</td>
<td>Blocks execution of the thread until one or more attached Conditions becomes true, or until a user-specified timeout expires. See 4.6.3 Waiting for Conditions on the next page.</td>
</tr>
</tbody>
</table>
Table 4.7 WaitSet Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dispatch</td>
<td>(Modern C++ API only) Blocks execution of the thread until one or more attached Conditions becomes true, or until a user-specified timeout expires. Then it calls the handlers attached to the active conditions and returns. For more information see the API Reference HTML documentation for the DDS Modern C++ API (Modules, Infrastructure Module, Conditions and WaitSets).</td>
</tr>
<tr>
<td>get_conditions</td>
<td>Retrieves a list of attached Conditions.</td>
</tr>
<tr>
<td>get_property</td>
<td>Retrieves the DDS_WaitSetProperty_t structure of the associated WaitSet.</td>
</tr>
<tr>
<td>set_property</td>
<td>Sets the DDS_WaitSetProperty_t structure, to configure the associated WaitSet to return after one or more trigger events have occurred.</td>
</tr>
</tbody>
</table>

### 4.6.3 Waiting for Conditions

The WaitSet’s wait() operation allows an application thread to wait for any of the attached Conditions to trigger (become TRUE).

If any of the attached Conditions are already TRUE when wait() is called, it returns immediately.

If none of the attached Conditions are already TRUE, wait() blocks—suspending the calling thread. The waiting behavior depends on whether or not properties were set when the WaitSet was created:

- **If properties are not specified when the WaitSet is created:**

  The WaitSet will wake up as soon as a trigger event occurs (that is, when an attached Condition becomes true). This is the default behavior if properties are not specified.

  This 'immediate wake-up' behavior is optimal if you want to minimize latency (to wake up and process the data or event as soon as possible). However, "waking up" involves a context switch—the operating system must signal and schedule the thread that is waiting on the WaitSet. A context switch consumes significant CPU and therefore waking up on each data update is not optimal in situations where the application needs to maximize throughput (the number of messages processed per second). This is especially true if the receiver is CPU limited.

- **If properties are specified when the WaitSet is created:**

  The properties configure the waiting behavior of a WaitSet. If no conditions are true at the time of the call to wait, the WaitSet will wait for (a) max_event_count trigger events to occur, (b) up to max_event_delay time from the occurrence of the first trigger event, or (c) up to the timeout maximum wait duration specified in the call to wait().  (**Note:** The resolution of the timeout period is constrained by the resolution of the system clock.)
4.6.3 Waiting for Conditions

If `wait()` does not timeout, it returns a list of the attached `Conditions` that became TRUE and therefore unblocked the wait.

If `wait()` does timeout, it returns TIMEOUT and an empty list of `Conditions`.

Only one application thread can be waiting on the same `WaitSet`. If `wait()` is called on a `WaitSet` that already has a thread blocking on it, the operation will immediately return PRECONDITION_NOT_MET.

If you detach a `Condition` from a `Waitset` that is currently in a wait state (that is, you are waiting on it), `wait()` may return OK and an empty sequence of conditions.

4.6.3.1 How WaitSets Block

The blocking behavior of the `WaitSet` is illustrated in Figure 4.8: WaitSet Blocking Behavior below. The result of a `wait()` operation depends on the state of the `WaitSet`, which in turn depends on whether at least one attached `Condition` has a `trigger_value` of TRUE.

If the `wait()` operation is called on a `WaitSet` with state BLOCKED, it will block the calling thread. If `wait()` is called on a `WaitSet` with state UNBLOCKED, it will return immediately.

When the `WaitSet` transitions from BLOCKED to UNBLOCKED, it wakes up the thread (if there is one) that had called `wait()` on it. There is no implied “event queuing” in the awakening of a `WaitSet`. That is, if several `Conditions` attached to the `WaitSet` have their `trigger_value` transition to true in sequence, Connext DDS will only unblock the `WaitSet` once.

Figure 4.8: WaitSet Blocking Behavior
4.6.4 Processing Triggered Conditions—What to do when Wait() Returns

When `wait()` returns, it provides a list of the attached `Condition` objects that have a `trigger_value` of true. Your application can use this list to do the following for each `Condition` in the returned list:

- If it is a `StatusCondition`:
  - First, call `get_status_changes()` to see what status changed.
  - If the status changes refer to plain communication status: call `get_<communication_status>()` on the relevant `Entity`.
  - If the status changes refer to `DATA_ON_READERS`:\[1\]: call `get_datareaders()` on the relevant `Subscriber`.
  - If the status changes refer to `DATAAVAILABLE`: call `read()` or `take()` on the relevant `DataReader`.

- If it is a `ReadCondition` or a `QueryCondition`: You may want to call `read_w_condition()` or `take_w_condition()` on the `DataReader`, with the `ReadCondition` as a parameter (see 8.4.3.6 `read_w_condition` and `take_w_condition` on page 561).

Note that this is just a suggestion, you do not have to use the “w_condition” operations (or any read/-take operations, for that matter) simply because you used a `WaitSet`. The “w_condition” operations are just a convenient way to use the same status masks that were set on the `ReadCondition` or `QueryCondition`.

- If it is a `GuardCondition`: check to see which `GuardCondition` changed, then react accordingly. Recall that `GuardConditions` are completely controlled by your application.

See 4.6.5 `Conditions and WaitSet Example` below to see how to determine which of the attached `Conditions` is in the returned list.

### 4.6.5 Conditions and WaitSet Example

This example creates a `WaitSet` and then waits for one or more attached `Conditions` to become true.

```java
// Create a WaitSet
WaitSet* waitset = new WaitSet();

// Attach Conditions
DDSCondition* cond1 = ...;
DDSCondition* cond2 = entity->get_statuscondition();
DDSCondition* cond3 = reader->create_readcondition(
    DDS_NOT_READ_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE);
DDSCondition* cond4 = new DDSGuardCondition();
DDSCondition* cond5 = ...;
```

1. And then read/take on the returned `DataReader` objects.
4.6.5 Conditions and WaitSet Example

```c
DDS_ReturnCode_t retcode;

retcode = waitset->attach_condition(cond1);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond2);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond3);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond4);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
retcode = waitset->attach_condition(cond5);
if (retcode != DDS_RETCODE_OK) {
    // ... error
}
// Wait for a condition to trigger or timeout
DDS_Duration_t timeout = { 0, 1000000 }; // 1ms
DDSConditionSeq active_conditions; // holder for active conditions
bool is_cond1_triggered = false;
bool is_cond2_triggered = false;
DDS_ReturnCode_t retcode;

retcode = waitset->wait(active_conditions, timeout);
if (retcode == DDS_RETCODE_TIMEOUT) {
    // handle timeout
    printf("Wait timed out. No conditions were triggered.\n");
} else if (retcode != DDS_RETCODE_OK) {
    // ... check for cause of failure
} else {
    // success
    if (active_conditions.length() == 0) {
        printf("Wait timed out!! No conditions triggered.\n");
    } else {
        // check if "cond1" or "cond2" are triggered:
        for(i = 0; i < active_conditions.length(); ++i) {
            if (active_conditions[i] == cond1) {
                printf("Cond1 was triggered!");
                is_cond1_triggered = true;
            }
            if (active_conditions[i] == cond2) {
                printf("Cond2 was triggered!");
                is_cond2_triggered = true;
            }
            if (is_cond1_triggered && is_cond2_triggered) {
                break;
            }
        }
    }
}
```
4.6.6 GuardConditions

GuardConditions are created by your application. GuardConditions provide a way for your application to manually awaken a WaitSet. Like all Conditions, it has a single boolean trigger_value. Your application can manually trigger the GuardCondition by calling set_trigger_value().

Connext DDS does not trigger or clear this type of condition—it is completely controlled by your application.

A GuardCondition has no factory. It is created as an object directly by the natural means in each language binding (e.g., using “new” in C++ or Java). For example:

```c++
// Create a Guard Condition
Condition* my_guard_condition = new GuardCondition();
// Delete a Guard Condition
delete my_guard_condition;
```

When first created, the trigger_value is FALSE.

A GuardCondition has only two operations, get_trigger_value() and set_trigger_value().

When your application calls set_trigger_value(DDS_BOOLEAN_TRUE), Connext DDS will awaken any WaitSet to which the GuardCondition is attached.

4.6.7 ReadConditions and QueryConditions

ReadConditions are created by your application, but triggered by Connext DDS. ReadConditions provide a way for you to specify the DDS data samples that you want to wait for, by indicating the desired sample-states, view-states, and instance-states (see 6.1 Instance States on page 258). Then Connext DDS will trigger the ReadCondition when suitable DDS samples are available.

A QueryCondition is a special ReadCondition that allows you to specify a query expression and parameters, so you can filter on the locally available (already received) data. QueryConditions use the same SQL-based filtering syntax as ContentFilteredTopics for query expressions, parameters, etc. Unlike ContentFilteredTopics, QueryConditions are applied to data already received, so they do not affect the reception of data.

Multiple mask combinations can be associated with a single content filter. This is important because the maximum number of content filters that may be created per DataReader is 32, but more than 32
QueryConditions may be created per DataReader, if they are different mask-combinations of the same content filter.

ReadConditions and QueryConditions are created by using the DataReader’s create_readcondition() and create_querycondition() operations. For example:

```c
DDSReadCondition* my_read_condition = 
    reader->create_readcondition( 
        DDS_NOT_READ_SAMPLE_STATE, 
        DDS_ANY_VIEW_STATE, 
        DDS_ANY_INSTANCE_STATE); 
DDSQueryCondition* my_query_condition = 
    reader->create_querycondition( 
        DDS_NOT_READ_SAMPLE_STATE, 
        DDS_ANY_VIEW_STATE, 
        DDS_ANY_INSTANCE_STATE 
        query_expression, 
        query_parameters); 
```

You can also use the alternative DataReader operations, create_readcondition_w_params() and create_querycondition_w_params(), which perform the same action as create_readcondition() and create_querycondition(), but allow the application to explicitly set the masks in the DDS_ReadConditionParams and DDS_QueryConditionParams structures (see Table 4.9 DDS_ReadConditionParams and Table 4.10 DDS_QueryConditionParams).

In addition, create_readcondition_w_params() and create_querycondition_w_params() allow selecting between TopicQuery samples and LIVE samples (see Topic Queries (Chapter 24 on page 977)).

A DataReader can have multiple attached ReadConditions and QueryConditions. A ReadCondition or QueryCondition may only be attached to one DataReader.

To delete a ReadCondition or QueryCondition, use the DataReader’s delete_readcondition() operation:

```c
DDS_ReturnCode_t delete_readcondition (DDSReadCondition *condition) 
```

After a ReadCondition is triggered, use the FooDataReader’s read/take “with condition” operations (see 8.4.3.6 read_w_condition and take_w_condition on page 561) to access the DDS samples.

Table 4.8 ReadCondition and QueryCondition Operations lists the operations available on ReadConditions.

### Table 4.8 ReadCondition and QueryCondition Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_datareader</td>
<td>Returns the DataReader to which the ReadCondition or QueryCondition is attached.</td>
</tr>
<tr>
<td>get_instance_state_mask</td>
<td>Returns the instance states that were specified when the ReadCondition or QueryCondition was created. These are the DDS sample’s instance states that Connext DDS checks to determine whether or not to trigger the ReadCondition or QueryCondition.</td>
</tr>
</tbody>
</table>
### 4.6.7 ReadConditions and QueryConditions

**Table 4.8 ReadCondition and QueryCondition Operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_sample_state_mask</td>
<td>Returns the sample-states that were specified when the ReadCondition or QueryCondition was created. These are the sample states that Connext DDS checks to determine whether or not to trigger the ReadCondition or QueryCondition.</td>
</tr>
<tr>
<td>get_view_state_mask</td>
<td>Returns the view-states that were specified when the ReadCondition or QueryCondition was created. These are the view states that Connext DDS checks to determine whether or not to trigger the ReadCondition or QueryCondition.</td>
</tr>
<tr>
<td>get_stream_kind_mask</td>
<td>Retrieves the stream kind mask for the condition.</td>
</tr>
</tbody>
</table>

**Table 4.9 DDS_ReadConditionParams**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_SampleStateMask</td>
<td>sample_states</td>
<td>Sample state of the data samples that are of interest.</td>
</tr>
<tr>
<td>DDS_ViewStateMask</td>
<td>view_states</td>
<td>View state of the data samples that are of interest.</td>
</tr>
<tr>
<td>DDS_InstanceStateMask</td>
<td>instance_states</td>
<td>Instance state of the data samples that are of interest.</td>
</tr>
<tr>
<td>DDS_StreamKindMask</td>
<td>stream_kinds</td>
<td>Stream kind of the data samples that are of interest.</td>
</tr>
</tbody>
</table>

**Table 4.10 DDS_QueryConditionParams**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct DDS_ReadConditionParams</td>
<td>as_readconditionparam</td>
<td>Read condition parameters</td>
</tr>
<tr>
<td>char *</td>
<td>query_expression</td>
<td>Expression for the query.</td>
</tr>
<tr>
<td>struct DDS_StringSeq</td>
<td>query_parameters</td>
<td>Parameters for the query expression.</td>
</tr>
</tbody>
</table>

### 4.6.7.1 How ReadConditions are Triggered

A ReadCondition has a **trigger value** that determines whether the attached WaitSet is BLOCKED or UNBLOCKED. Unlike the StatusCondition, the trigger value of the ReadCondition is tied to the presence of at least one DDS sample with a sample-state, view-state, and instance-state that matches those set in the ReadCondition. Furthermore, for the QueryCondition to have a trigger value==TRUE, the data associated with the DDS sample must be such that the query_expression evaluates to TRUE.

The trigger value of a ReadCondition depends on the presence of DDS samples on the associated DataReader. This implies that a single ‘take’ operation can potentially change the trigger value of several ReadConditions or QueryConditions. For example, if all DDS samples are taken, any ReadConditions and
QueryConditions associated with the DataReader that had trigger_value==TRUE before will see the trigger_value change to FALSE. Note that this does not guarantee that WaitSet objects that were separately attached to those conditions will not be awakened. Once we have trigger_value==TRUE on a condition, it may wake up the attached WaitSet, the condition transitioning to trigger_value==FALSE does not necessarily 'unwakeup' the WaitSet, since 'unwakening' may not be possible. The consequence is that an application blocked on a WaitSet may return from wait() with a list of conditions, some of which are no longer “active.” This is unavoidable if multiple threads are concurrently waiting on separate WaitSet objects and taking data associated with the same DataReader.

Consider the following example: A ReadCondition that has a sample_state_mask = {NOT_READ} will have a trigger_value of TRUE whenever a new DDS sample arrives and will transition to FALSE as soon as all the newly arrived DDS samples are either read (so their status changes to READ) or taken (so they are no longer managed by Connext DDS). However, if the same ReadCondition had a sample_state_mask = {READ, NOT_READ}, then the trigger_value would only become FALSE once all the newly arrived DDS samples are taken (it is not sufficient to just read them, since that would only change the SampleState to READ), which overlaps the mask on the ReadCondition.

### 4.6.7.2 QueryConditions

A QueryCondition is a special ReadCondition that allows your application to also specify a filter on the locally available data.

The query expression is similar to a SQL WHERE clause and can be parameterized by arguments that are dynamically changeable by the set_query_parameters() operation.

QueryConditions are triggered in the same manner as ReadConditions, with the additional requirement that the DDS sample must also satisfy the conditions of the content filter associated with the QueryCondition.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_query_expression</td>
<td>Returns the query expression specified when the QueryCondition was created.</td>
</tr>
<tr>
<td>get_query_parameters</td>
<td>Returns the query parameters associated with the QueryCondition. That is, the parameters specified on the last successful call to set_query_parameters(), or if set_query_parameters() was never called, the arguments specified when the QueryCondition was created.</td>
</tr>
<tr>
<td>set_query_parameters</td>
<td>Changes the query parameters associated with the QueryCondition.</td>
</tr>
</tbody>
</table>

### 4.6.8 StatusConditions

StatusConditions are created automatically by Connext DDS, one for each Entity. Connext DDS will trigger the StatusCondition when there is a change to any of that Entity’s enabled statuses.
By default, when Connext DDS creates a StatusCondition, all status bits are turned on, which means it will check for all statuses to determine when to trigger the StatusCondition. If you only want Connext DDS to check for specific statuses, you can use the StatusCondition’s set_enabled_statuses() operation and set just the desired status bits.

The trigger_value of the StatusCondition depends on the communication status of the Entity (e.g., arrival of data, loss of information, etc.), ‘filtered’ by the set of enabled statuses on the StatusCondition.

The set of enabled statuses and its relation to Listeners and WaitSets is detailed in 4.6.8.1 How StatusConditions are Triggered below.

Table 4.12 StatusCondition Operations lists the operations available on StatusConditions.

### Table 4.12 StatusCondition Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_enabled_statuses</td>
<td>Defines the list of communication statuses that are taken into account to determine the trigger_value of the StatusCondition. This operation may change the trigger_value of the StatusCondition. WaitSets behavior depend on the changes of the trigger_value of their attached conditions. Therefore, any WaitSet to which the StatusCondition is attached is potentially affected by this operation. If this function is not invoked, the default list of enabled statuses includes all the statuses.</td>
</tr>
<tr>
<td>get_enabled_statuses</td>
<td>Retrieves the list of communication statuses that are taken into account to determine the trigger_value of the StatusCondition. This operation returns the statuses that were explicitly set on the last call to set_enabled_statuses() or, if set_enabled_statuses() was never called, the default list</td>
</tr>
<tr>
<td>get_entity</td>
<td>Returns the Entity associated with the StatusCondition. Note that there is exactly one Entity associated with each StatusCondition.</td>
</tr>
</tbody>
</table>

Unlike other types of Conditions, StatusConditions are created by Connext DDS, not by your application. To access an Entity’s StatusCondition, use the Entity’s get_statuscondition() operation. For example:

```
Condition* my_status_condition = entity->get_statuscondition();
```

In the Modern C++ API, use the StatusCondition constructor to obtain a reference to the Entity’s condition. For example:

```
dds::core::cond::StatusCondition my_status_condition(entity)
```

After a StatusCondition is triggered, call the Entity’s get_status_changes() operation to see which status (es) changed.

Note: Not all statuses will activate the StatusCondition. Refer to the API Reference HTML documentation of the individual statuses for that information.

### 4.6.8.1 How StatusConditions are Triggered

The trigger_value of a StatusCondition is the boolean OR of the ChangedStatusFlag of all the communication statuses to which it is sensitive. That is, trigger_value is FALSE only if all the values of the ChangedStatusFlags are FALSE.
The sensitivity of the \textit{StatusCondition} to a particular communication status is controlled by the list of \texttt{enabled_statuses} set on the \textit{Condition} by means of the \texttt{set_enabled_statuses()} operation.

Once a \textit{StatusCondition}'s \texttt{trigger_value} becomes true, it remains true until the status that changed is reset. To reset a status, call the related \texttt{get_*_status()} operation. Or, in the case of the data available status, call \texttt{read()}, \texttt{take()}, or one of their variants.

Therefore, if you are using a \textit{StatusCondition} on a \textit{WaitSet} to be notified of events, your thread will wake up when one of the statuses associated with the \textit{StatusCondition} becomes true. If you do not reset the status, the \textit{StatusCondition}'s \texttt{trigger_value} remains true and your \textit{WaitSet} will not block again—it will immediately wake up when you call \texttt{wait()}.  

\textbf{4.6.9 Using Both Listeners and WaitSets}

You can use \textit{Listeners} and \textit{WaitSets} in the same application. For example, you may want to use \textit{WaitSets} and \textit{Conditions} to access the data, and \textit{Listeners} to be warned asynchronously of erroneous communication statuses.

We recommend that you choose one or the other mechanism for each particular communication status (not both). However, if both are enabled, the \textit{Listener} mechanism is used first, then the \textit{WaitSet} objects are signaled.
Chapter 5 Working with Topics

For a DataWriter and DataReader to communicate, they need to use the same Topic. A Topic includes a name and an association with a user data type that has been registered with Connext DDS. Topic names are how different parts of the communication system find each other. Topics are named streams of data of the same data type. DataWriters publish DDS samples into the stream; DataReaders subscribe to data from the stream. More than one Topic can use the same user data type, but each Topic needs a unique name.

Topics, DataWriters, and DataReaders relate to each other as follows:

- Multiple Topics (each with a unique name) can use the same user data type.
- Applications may have multiple DataWriters for each Topic.
- Applications may have multiple DataReaders for each Topic.
- DataWriters and DataReaders must be associated with the same Topic in order for them to be connected.
- Topics are created and deleted by a DomainParticipant, and as such, are owned by that DomainParticipant. When two applications (DomainParticipants) want to use the same Topic, they must both create the Topic (even if the applications are on the same node).

Connext DDS uses ‘Builtin Topics’ to discover and keep track of remote entities, such as new participants in the DDS domain. Builtin Topics are discussed in Built-In Topics (Chapter 18 on page 831).

5.1 Topics

Before you can create a Topic, you need a user data type (see Data Types and DDS Data Samples (Chapter 3 on page 27)) and a DomainParticipant (DomainParticipants (9.3 on page 609)). The user data type must be registered with the DomainParticipant (see 3.2.8 Type Codes for Built-in Types on page 74).
5.1 Topics

Once you have created a *Topic*, what do you do with it? Topics are primarily used as parameters in other *Entities’* operations. For instance, a Topic is required when a *Publisher* or *Subscriber* creates a *DataWriter* or *DataReader*, respectively. *Topics* do have a few operations of their own, as listed in Table 5.1 *Topic Operations*. For details on using these operations, see the reference section or the API Reference HTML documentation.

Figure 5.1: Topic Module
Table 5.1 Topic Operations

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuring the Topic</td>
<td>enable</td>
<td>Enables the Topic.</td>
<td>4.1.2 Enabling DDS Entities on page 168</td>
</tr>
<tr>
<td></td>
<td>get_qos</td>
<td>Gets the Topic's current QosPolicy settings. This is most often used in preparation for calling set_qos().</td>
<td>5.1.3 Setting Topic QosPolicies on page 219</td>
</tr>
<tr>
<td></td>
<td>set_qos</td>
<td>Sets the Topic's QoS. You can use this operation to change the values for the Topic's QosPolicies. Note, however, that not all QosPolicies can be changed after the Topic has been created.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Compares two Topic's QoS structures for equality.</td>
<td>5.1.3.2 Comparing QoS Values on page 221</td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>Sets the Topic's QoS based on a specified QoS profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_listener</td>
<td>Gets the currently installed Listener.</td>
<td>5.1.5 Setting Up Top- icListeners on page 223</td>
</tr>
<tr>
<td></td>
<td>set_listener</td>
<td>Sets the Topic's Listener. If you create the Topic without a Listener, you can use this operation to add one later. Setting the listener to NULL will remove the listener from the Topic.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>narrow</td>
<td>A type-safe way to cast a pointer. This takes a DDSTopicDescription pointer and 'narrows' it to a DDSTopic pointer.</td>
<td>7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308</td>
</tr>
<tr>
<td>Checking Status</td>
<td>get_inconsistent_topic_status</td>
<td>Allows an application to retrieve a Topic's INCONSISTENT_TOPIC_STATUS status.</td>
<td>5.3.1 INCONSISTENT_TOPIC Status on page 226</td>
</tr>
<tr>
<td></td>
<td>get_status_changes</td>
<td>Gets a list of statuses that have changed since the last time the application read the status or the listeners were called.</td>
<td>4.1.4 Getting Status and Status Changes on page 171</td>
</tr>
<tr>
<td>Navigating Relationships</td>
<td>get_name</td>
<td>Gets the topic_name string used to create the Topic.</td>
<td>5.1.1 Creating Topics below</td>
</tr>
<tr>
<td></td>
<td>get_type_name</td>
<td>Gets the type_name used to create the Topic.</td>
<td>5.1.6.1 Finding a Topic's DomainParticipant on page 223</td>
</tr>
<tr>
<td></td>
<td>get_participant</td>
<td>Gets the DomainParticipant to which this Topic belongs.</td>
<td></td>
</tr>
</tbody>
</table>

5.1.1 Creating Topics

Topics are created using the DomainParticipant's create_topic() or create_topic_with_profile() operation.

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see Configuring QoS with XML (Chapter 19 on page 848).
5.1.1 Creating Topics

```
DDSTopic * create_topic (  
    const char *topic_name,  
    const char *type_name,  
    const DDS_TOPICQos &qos,  
    DDSTopicListener *listener,  
    DDS_StatusMask mask)
```

```
DDSTopic * create_topic_with_profile (  
    const char *topic_name,  
    const char *type_name,  
    const char *library_name,  
    const char *profile_name,  
    DDSTopicListener *listener,  
    DDS_StatusMask mask)
```

Where:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>topic_name</td>
<td>Name for the new Topic, must not exceed 255 characters.</td>
</tr>
<tr>
<td>type_name</td>
<td>Name for the user data type, must not exceed 255 characters. It must be the same name that was used to register the DDS type, and the DDS type must be registered with the same DomainParticipant used to create this Topic. See 3.6 Using RTI Code Generator (rtdsgen) on page 152.</td>
</tr>
<tr>
<td>qos</td>
<td>If you want to use the default QoS settings (described in the API Reference HTML documentation), use DDS_TOPIC_QOS_DEFAULT for this parameter (see Figure 5.2: Creating a Topic with Default QosPolicies on the next page). If you want to customize any of the QosPolicies, supply a QoS structure (see 5.1.3 Setting Topic QosPolicies on the next page). If you use DDS_TOPIC_QOS_DEFAULT, it is not safe to create the topic while another thread may be simultaneously calling the DomainParticipant's set_default_topic_qos() operation.</td>
</tr>
<tr>
<td>listener</td>
<td>Listeners are callback routines. Connext DDS uses them to notify your application of specific events (status changes) that may occur with respect to the Topic. The listener parameter may be set to NULL if you do not want to install a Listener. If you use NULL, the Listener of the DomainParticipant to which the Topic belongs will be used instead (if it is set). For more information on TopicListeners, see 5.1.5 Setting Up TopicListeners on page 223.</td>
</tr>
<tr>
<td>mask</td>
<td>This bit-mask indicates which status changes will cause the Listener to be invoked. The bits in the mask that are set must have corresponding callbacks implemented in the Listener. If you use NULL for the Listener, use DDS_STATUS_MASK_NONE for this parameter. If the Listener implements all callbacks, use DDS_STATUS_MASK_ALL. For information on statuses, see 4.4 Listeners on page 189.</td>
</tr>
<tr>
<td>library_name</td>
<td>A QoS Library is a named set of QoS profiles. See 19.3 QoS Profiles on page 850. If NULL is used for library_name, the DomainParticipant's default library is assumed.</td>
</tr>
<tr>
<td>profile_name</td>
<td>A QoS profile groups a set of related QoS, usually one per entity. See 19.3 QoS Profiles on page 850. If NULL is used for profile_name, the DomainParticipant's default profile is assumed and library_name is ignored.</td>
</tr>
</tbody>
</table>

It is not safe to create a topic while another thread is calling lookup_topicdescription() for that same topic (see 9.3.8 Looking up Topic Descriptions on page 628).
5.1.2 Deleting Topics

To delete a Topic, use the DomainParticipant’s delete_topic() operation:

```c
DDS_ReturnCode_t delete_topic (DDSTopic * topic)
```

Note, however, that you cannot delete a Topic if there are any existing DataReaders or DataWriters (belonging to the same DomainParticipant) that are still using it. All DataReaders and DataWriters associated with the Topic must be deleted first.

**Note:** In the Modern C++ API, Entities are automatically destroyed.

5.1.3 Setting Topic QosPolicies

A Topic’s QosPolicies control its behavior, or more specifically, the behavior of the DataWriters and DataReaders of the Topic. You can think of the policies as the ‘properties’ for the Topic. The DDS_TopicQos structure has the following format:

```c
DDS_TopicQos struct {
    DDS_TopicDataQosPolicy topic_data;
    DDS_DurabilityQosPolicy durability;
    DDS_DurabilityServiceQosPolicy durability_service;
    DDS_DeadlineQosPolicy deadline;
    DDS_LatencyBudgetQosPolicy latency_budget;
    DDS_LivelinessQosPolicy liveliness;
    DDS_ReliabilityQosPolicy reliability;
    DDS_DestinationOrderQosPolicy destination_order;
    DDS_HistoryQosPolicy history;
    DDS_ResourceLimitsQosPolicy resource_limits;
    DDS_TransportPriorityQosPolicy transport_priority;
    DDS_LifespanQosPolicy lifespan;
    DDS_OwnershipQosPolicy ownership;
}
```
Table 5.2 Topic QosPolicies summarizes the meaning of each policy (arranged alphabetically). For information on why you would want to change a particular QosPolicy, see the section noted in the Reference column. For defaults and valid ranges, please refer to the API Reference HTML documentation for each policy.

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataRepresentation</td>
<td>Specifies which versions of the Extended Common Data Representation (CDR) are offered and requested. See 7.5.3 DATA_REPRESENTATION QosPolicy on page 380. During Publisher_copy_from_topic_qos, only the first DataRepresentation.node.element is copied to the DataWriterQos. The whole sequence is copied to the DataReaderQos during Subscriber_copy_from_topic_qos.</td>
</tr>
<tr>
<td>Deadline</td>
<td>For a DataReader, specifies the maximum expected elapsed time between arriving DDS data samples. For a DataWriter, specifies a commitment to publish DDS samples with no greater elapsed time between them. See 7.5.7 DEADLINE QosPolicy on page 405.</td>
</tr>
<tr>
<td>DestinationOrder</td>
<td>Controls how Connext DDS will deal with data sent by multiple DataWriters for the same topic. Can be set to &quot;by reception timestamp&quot; or to &quot;by source timestamp&quot;. See 7.5.8 DESTINATION_ORDER QosPolicy on page 408.</td>
</tr>
<tr>
<td>Durability</td>
<td>Specifies whether or not Connext DDS will store and deliver data that were previously published to new DataReaders. See 7.5.9 DURABILITY QosPolicy on page 411.</td>
</tr>
<tr>
<td>DurabilityService</td>
<td>Various settings to configure the external Persistence Service used by Connext DDS for DataWriters with a Durability QoS setting of Persistent Durability. See 7.5.10 DURABILITY SERVICE QosPolicy on page 416.</td>
</tr>
<tr>
<td>History</td>
<td>Specifies how much data must be stored by Connext DDS for the DataWriter or DataReader. This QosPolicy affects the 7.5.21 RELIABILITY QosPolicy on page 447 as well as the 7.5.9 DURABILITY QosPolicy on page 411. See 7.5.12 HISTORY QosPolicy on page 420.</td>
</tr>
<tr>
<td>LatencyBudget</td>
<td>Suggestion to Connext DDS on how much time is allowed to deliver data. See 7.5.13 LATENCYBUDGET QoS Policy on page 425.</td>
</tr>
<tr>
<td>Lifespan</td>
<td>Specifies how long Connext DDS should consider data sent by an user application to be valid. See 7.5.14 LIFESPAN QoS Policy on page 425.</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Specifies and configures the mechanism that allows DataReaders to detect when DataWriters become disconnected or &quot;dead.&quot; See 7.5.15 LIVELINESS QosPolicy on page 427.</td>
</tr>
<tr>
<td>Ownership</td>
<td>Along with Ownership Strength, specifies if DataReaders for a topic can receive data from multiple DataWriters at the same time. See 7.5.17 OWNERSHIP QosPolicy on page 434.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Specifies whether or not Connext DDS will deliver data reliably. See 7.5.21 RELIABILITY QosPolicy on page 447.</td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>Controls the amount of physical memory allocated for entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.</td>
</tr>
<tr>
<td>TopicData</td>
<td>Along with Group Data QosPolicy and User Data QosPolicy, used to attach a buffer of bytes to Connext DDS's discovery meta-data. See 5.2.1 TOPIC_DATA QosPolicy on page 224.</td>
</tr>
</tbody>
</table>
### 5.1.3 Setting Topic QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TransportPriority</td>
<td>Set by a DataWriter to tell Connext DDS that the data being sent is a different &quot;priority&quot; than other data. See 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458.</td>
</tr>
</tbody>
</table>

#### 5.1.3.1 Configuring QoS Settings when the Topic is Created

As described in 5.1.1 Creating Topics on page 217, there are different ways to create a Topic, depending on how you want to specify its QoS (with or without a QoS profile).

In Figure 5.2: Creating a Topic with Default QosPolicies on page 219, we saw an example of how to create a Topic with default QosPolicies by using the special constant, DDS_TOPIC_QOS_DEFAULT, which indicates that the default QoS values for a Topic should be used. The default Topic QoS values are configured in the DomainParticipant; you can change them with the DomainParticipant’s set_default_topic_qos() or set_default_topic_qos_with_profile() operations (see 9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627).

To create a Topic with non-default QoS values, without using a QoS profile, use the DomainParticipant’s get_default_topic_qos() operation to initialize a DDS_TopicQos structure. Then change the policies from their default values before passing the QoS structure to create_topic().

You can also create a Topic and specify its QoS settings via a QoS profile. To do so, call create_topic_with_profile().

If you want to use a QoS profile, but then make some changes to the QoS before creating the Topic, call get_topic_qos_from_profile(), modify the QoS and use the modified QoS when calling create_topic().

#### 5.1.3.2 Comparing QoS Values

The equals() operation compares two Topic’s DDS_TopicQos structures for equality. It takes two parameters for the two Topics’ QoS structures to be compared, then returns TRUE if they are equal (all values are the same) or FALSE if they are not equal.

#### 5.1.3.3 Changing QoS Settings After the Topic Has Been Created

There are two ways to change an existing Topic’s QoS after it is has been created—again depending on whether or not you are using a QoS Profile.

To change QoS programmatically (that is, without using a QoS Profile), see the example code in Figure 5.3: Changing the QoS of an Existing Topic (without a QoS Profile) on the next page. It retrieves the current values by calling the Topic’s get_qos() operation. Then it modifies the value and calls set_qos() to apply the new value. Note, however, that some QosPolicies cannot be changed after the Topic has been enabled—this restriction is noted in the descriptions of the individual QosPolicies.
5.1.4 Copying QoS From a Topic to a DataWriter or DataReader

You can also change a Topic’s (and all other Entities’) QoS by using a QoS Profile. For an example, see Figure 5.4: Changing the QoS of an Existing Topic with a QoS Profile below. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

For the C API, use DDS_TopicQos_INITIALIZER or DDS_TopicQos_initialize(). See 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

Figure 5.3: Changing the QoS of an Existing Topic (without a QoS Profile)

```c
DDS_TopicQos topic_qos;
// Get current QoS. topic points to an existing DDSTopic.
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Next, make changes.
// New ownership kind will be Exclusive
topic_qos.ownership.kind = DDS_EXCLUSIVE_OWNERSHIP_QOS;
// Set the new QoS
if (topic->set_qos(topic_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

Figure 5.4: Changing the QoS of an Existing Topic with a QoS Profile

```c
retcode = topic->set_qos_with_profile(
    "FooProfileLibrary","FooProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

5.1.4 Copying QoS From a Topic to a DataWriter or DataReader

Only the TOPIC_DATA QosPolicy strictly applies to Topics—it is described in this section, while the others are described in the sections noted Table 5.2 Topic QosPolicies. The rest of the QosPolicies for a Topic can also be set on the corresponding DataWriters and/or DataReaders. Actually, the values that Connext DDS uses for those policies are taken directly from those set on the DataWriters and DataReaders. The values for those policies are stored only for reference in the DDS_TopicQos structure.

Because many QosPolicies affect the behavior of matching DataWriters and DataReaders, the DDS_TopicQos structure is provided as a convenient way to set the values for those policies in a single place in the application. Otherwise, you would have to modify the individual QosPolicies within separate DataWriter and DataReader QoS structures. And because some QosPolicies are compared between DataReaders and DataWriters, you will need to make certain that the individual values that you set are compatible (see 4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property on page 180).

The use of the DDS_TopicQos structure to set the values of any QosPolicy except TOPIC_DATA—which only applies to Topics—is really a way to share a single set of values with the associated DataWriters and DataReaders, as well as to avoid creating those entities with inconsistent QosPolicies.

To cause a DataWriter to use its Topic’s QoS settings, either:
5.1.5 Setting Up TopicListeners

- Pass DDS_DATAWRITER_QOS_USE_TOPIC_QOS to `create_datawriter()`, or
- Call the Publisher’s `copy_from_topic_qos()` operation

To cause a `DataReader` to use its `Topic`’s QoS settings, either:

- Pass DDS_DATAREADER_QOS_USE_TOPIC_QOS to `create_datareader()`, or
- Call the Subscriber’s `copy_from_topic_qos()` operation

Please refer to the API Reference HTML documentation for the Publisher’s `create_datawriter()` and Subscriber’s `create_datareader()` methods for more information about using values from the `Topic` QosPolicies when creating `DataWriters` and `DataReaders`.

5.1.5 Setting Up TopicListeners

When you create a `Topic`, you have the option of giving it a `Listener`. A `TopicListener` includes just one callback routine, `on_inconsistent_topic()`. If you create a `TopicListener` (either as part of the `Topic` creation call, or later with the `set_listener()` operation), Connext DDS will invoke the `TopicListener’s on_inconsistent_topic()` method whenever it detects that another application has created a `Topic` with same name but associated with a different user data type. For more information, see 5.3.1 INCONSISTENT_TOPIC Status on page 226.

**Note:** Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

If a `Topic`’s Listener has not been set and Connext DDS detects an inconsistent `Topic`, the `DomainParticipantListener` (if it exists) will be notified instead (see 9.3.6 Setting Up DomainParticipantListeners on page 620). So you only need to set up a `TopicListener` if you need to perform specific actions when there is an error on that particular `Topic`. In most cases, you can set the `TopicListener` to NULL and process inconsistent-topic errors in the `DomainParticipantListener` instead.

5.1.6 Navigating Relationships Among Entities

5.1.6.1 Finding a Topic’s DomainParticipant

To retrieve a handle to the `Topic`’s `DomainParticipant`, use the `get_participant()` operation:

```c
DDSDomainParticipant* DDSTopicDescription::get_participant()
```

Notice that this method belongs to the `DDSTopicDescription` class, which is the base class for `DDSTopic`.

5.1.6.2 Retrieving a Topic’s Name or DDS Type Name

If you want to retrieve the `topic_name` or `type_name` used in the `create_topic()` operation, use these methods:
Notice that these methods belong to the `DDSTopicDescription` class, which is the base class for `DDSTopic`.

## 5.2 Topic QosPolicies

This section describes the only QosPolicy that strictly applies to *Topics* (and no other types of *Entities*)—the TOPIC_DATA QosPolicy. For a complete list of the QosPolicies that can be set for *Topics*, see Table 5.2 Topic QosPolicies.

Most of the QosPolicies that can be set on a *Topic* can also be set on the corresponding *DataWriter* and/or *DataReader*. The *Topic*'s QosPolicy is essentially just a place to store QoS settings that you plan to share with multiple entities that use that *Topic* (see how in 5.1.3 Setting Topic QosPolicies on page 219); they are not used otherwise and are not propagated on the wire.

### 5.2.1 TOPIC_DATA QosPolicy

This QosPolicy provides an area where your application can store additional information related to the *Topic*. This information is passed between applications during discovery (see Discovery (Chapter 15 on page 764)) using builtin-topics (see Built-In Topics (Chapter 18 on page 831)). How this information is used will be up to user code. Connext DDS does not do anything with the information stored as TOPIC_DATA except to pass it to other applications. Use cases are usually application-to-application identification, authentication, authorization, and encryption purposes.

The value of the TOPIC_DATA QosPolicy is sent to remote applications when they are first discovered, as well as when the *Topic*'s `set_qos()` method is called after changing the value of the TOPIC_DATA. User code can set listeners on the builtin *DataReaders* of the builtin *Topics* used by Connext DDS to propagate discovery information. Methods in the builtin topic listeners will be called whenever new applications, *DataReaders*, and *DataWriters* are found. Within the user callback, you will have access to the TOPIC_DATA that was set for the associated *Topic*.

Currently, TOPIC_DATA of the associated *Topic* is only propagated with the information that declares a *DataWriter* or *DataReader*. Thus, you will need to access the value of TOPIC_DATA through DDS_PUBLICATIONBUILTINTOPICDATA or DDS_SUBSCRIPTIONBUILTINTOPICDATA (see Built-In Topics (Chapter 18 on page 831)).

The structure for the TOPIC_DATA QosPolicy includes just one field, as seen in Table 5.3 DDS_TOPICDATAQosPolicy. The field is a sequence of octets that translates to a contiguous buffer of bytes whose contents and length is set by the user. The maximum size for the data are set in the 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654.
Table 5.3 DDS_TopicDataQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_OctetSeq</td>
<td>value</td>
<td>default: empty</td>
</tr>
</tbody>
</table>

This policy is similar to the GROUP_DATA (7.4.4 GROUP_DATA QosPolicy on page 353) and USER_DATA (7.5.30 USER_DATA QosPolicy on page 465) policies that apply to other types of Entities.

5.2.1.1 Example

One possible use of TOPIC_DATA is to send an associated XML schema that can be used to process the data stored in the associated user data structure of the Topic. The schema, which can be passed as a long sequence of characters, could be used by an XML parser to take DDS samples of the data received for a Topic and convert them for updating some graphical user interface, web application or database.

5.2.1.2 Properties

This QosPolicy can be modified at any time. A change in the QosPolicy will cause Connext DDS to send packets containing the new TOPIC_DATA to all of the other applications in the DDS domain.

Because Topics are created independently by the applications that use the Topic, there may be different instances of the same Topic (same topic name and DDS data type) in different applications. The TOPIC_DATA for different instances of the same Topic may be set differently by different applications.

5.2.1.3 Related QosPolicies

- 7.4.4 GROUP_DATA QosPolicy on page 353
- 7.5.30 USER_DATA QosPolicy on page 465
- 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654

5.2.1.4 Applicable DDS Entities

- 5.1 Topics on page 215

5.2.1.5 System Resource Considerations

As mentioned earlier, the maximum size of the TOPIC_DATA is set in the topic_data_max_length field of the 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654. Because Connext DDS will allocate memory based on this value, you should only increase this value if you need to. If your system does not use TOPIC_DATA, then you can set this value to 0 to save
memory. Setting the value of the TOPIC_DATA QosPolicy to hold data longer than the value set in the topic_data_max_length field will result in failure and an INCONSISTENT_QOS_POLICY return code.

However, should you decide to change the maximum size of TOPIC_DATA, you must make certain that all applications in the DDS domain have changed the value of topic_data_max_length to be the same. If two applications have different limits on the size of TOPIC_DATA, and one application sets the TOPIC_DATA QosPolicy to hold data that is greater than the maximum size set by another application, then the DataWriters and DataReaders of that Topic between the two applications will not connect. This is also true for the GROUP_DATA (7.4.4 GROUP_DATA QosPolicy on page 353) and USER_DATA (7.5.30 USER_DATA QosPolicy on page 465) QosPolicies.

5.3 Status Indicator for Topics

There is only one communication status defined for a Topic, ON_INCONSISTENT_TOPIC. You can use the get_inconsistent_topic_status() operation to access the current value of the status or use a TopicListener to catch the change in the status as it occurs. See 4.4 Listeners on page 189 for a general discussion on Listeners and Statuses.

5.3.1 INCONSISTENT_TOPIC Status

In order for a DataReader and a DataWriter with the same Topic to communicate, their DDS types must be consistent according to the DataReader’s type-consistency enforcement policy value, defined in its 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594). This status indicates that another DomainParticipant has created a Topic using the same name as the local Topic, but with an inconsistent DDS type.

The status is a structure of type DDS_InconsistentTopicStatus, see Table 5.4 DDS_InconsistentTopicStatus Structure. The total_count keeps track of the total number of (DataReader, DataWriter) pairs with topic names that match the Topic to which this status is attached, but whose DDS types are inconsistent. The TopicListener’s on_inconsistent_topic() operation is invoked when this status changes (an inconsistent topic is found). You can also retrieve the current value by calling the Topic’s get_inconsistent_topic_status() operation.

The value of total_count_change reflects the number of inconsistent topics that were found since the last time get_inconsistent_topic_status() was called by user code or on_inconsistent_topic() was invoked by Connext DDS.

### Table 5.4 DDS_InconsistentTopicStatus Structure

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Total cumulative count of (DataReader, DataWriter) pairs whose topic names match the Topic to which this status is attached, but whose DDS types are inconsistent.</td>
</tr>
</tbody>
</table>


### 5.4 ContentFilteredTopics

A ContentFilteredTopic is a *Topic* with filtering properties. It makes it possible to subscribe to topics and at the same time specify that you are only interested in a subset of the topic’s data.

For example, suppose you have a *Topic* that contains a temperature reading for a boiler, but you are only interested in temperatures outside the normal operating range. A ContentFilteredTopic can be used to limit the number of DDS data samples a *DataReader* has to process and may also reduce the amount of data sent over the network.

#### 5.4.1 Overview

A ContentFilteredTopic creates a relationship between a *Topic*, also called the related topic, and user-specified filtering properties. The filtering properties consist of an expression and a set of parameters.

- The filter expression evaluates a logical expression on the Topic content. The filter expression is similar to the WHERE clause in a SQL expression.
- The parameters are strings that give values to the 'parameters' in the filter expression. There must be one parameter string for each parameter in the filter expression.

A ContentFilteredTopic is a type of topic description, and can be used to create *DataReaders*. However, a ContentFilteredTopic is *not* an entity—it does not have QoS Policies or Listeners.

A ContentFilteredTopic relates to other entities in Connext DDS as follows:

- ContentFilteredTopics are used when creating *DataReaders*, not *DataWriters*.
- Multiple *DataReaders* can be created with the same ContentFilteredTopic.
- A ContentFilteredTopic belongs to (is created/deleted by) a *DomainParticipant*.
- A ContentFilteredTopic and *Topic* must be in the same *DomainParticipant*.
- A ContentFilteredTopic can only be related to a single *Topic*.
- A *Topic* can be related to multiple ContentFilteredTopics.
- A ContentFilteredTopic can have the same name as a *Topic*, but ContentFilteredTopics must have unique names within the same *DomainParticipant*.
- A *DataReader* created with a ContentFilteredTopic will use the related Topic's QoS and Listeners.

#### Table 5.4 DDS_InconsistentTopicStatus Structure

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count_change</td>
<td>The change in total_count since the last time this status was read.</td>
</tr>
</tbody>
</table>
5.4.2 Where Filtering is Applied—Publishing vs. Subscribing Side

- Changing filter parameters on a ContentFilteredTopic causes all DataReaders using the same ContentFilteredTopic to see the change.
- A Topic cannot be deleted as long as at least one ContentFilteredTopic that has been created with it exists.
- A ContentFilteredTopic cannot be deleted as long as at least one DataReader that has been created with the ContentFilteredTopic exists.

5.4.2 Where Filtering is Applied—Publishing vs. Subscribing Side

Filtering may be performed on either side of the distributed application. (The DataWriter obtains the filter expression and parameters from the DataReader during discovery.)

When batching is enabled, content filtering is always done on the reader side.

Connext DDS also supports network-switch filtering for multi-channel DataWriters (see Multi-channel DataWriters (Chapter 20 on page 896)).

A DataWriter will automatically filter DDS data samples for a DataReader if all of the following are true; otherwise filtering is performed by the DataReader.

1. The DataWriter is filtering for no more than writer_resource_limits.max_remote_reader_filters DataReaders at the same time.
   - There is a resource-limit on the DataWriter called writer_resource_limits.max_remote_reader_filters (see 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401). This value can be from [0, $(2^{31})-2$] or DDS_LENGTH_UNLIMITED (default). 0 means do not filter any DataReader; 1 to $(2^{31})-2$ means that the DataWriter will filter for up to the specified number of DataReaders, and the Datawriter will store the result of the filtering per sample per DataReader; DDS_LENGTH_UNLIMITED means that the DataWriter will filter for up to $(2^{31})-2$ DataReaders, but in this case the DataWriter will not store the filtering result per sample per DataReader: if a sample is resent (such as due to a loss of reliable communication), the sample will be filtered again.
   - If a DataWriter is filtering max_remote_reader_filters DataReaders at the same time and a new filtered DataReader is created, then the newly created DataReader (max_remote_reader_filters + 1) is not filtered. Even if one of the first (max_remote_reader_filters) DataReaders is deleted, that already created DataReader (max_remote_reader_filters + 1) will still not be filtered. However, any subsequently created DataReaders will be filtered as long as the number of DataReaders currently being filtered is not more than writer_resource_limits.max_remote_reader_filters.

2. The DataReader is not subscribing to data using multicast.

3. There are no more than four matching DataReaders in the same locator (transport destination, for example IP address + port).
5.4.3 Creating ContentFilteredTopics

**Note:** Connext DDS supports limited writer-side filtering if there are more than four matching DataReaders in the same locator. The middleware will not send any sample to a locator if the sample is filtered out by all the DataReaders receiving samples on that locator. However, if there is one DataReader to which the sample has to be sent, all the DataReaders on the locator will perform reader-side filtering for the incoming sample.

4. The DataWriter has infinite liveness. (See 7.5.15 LIVENESS QosPolicy on page 427.)

5. The DataWriter is not using an Asynchronous Publisher. (That is, the DataWriter’s PUBLISH_MODE QosPolicy (DDS Extension) on page 444 kind is set to DDS_SYNCHRONOUS_PUBLISHER_MODE_QOS.)

**Note:** Connext DDS supports limited writer-side filtering if asynchronous publishing is enabled. The middleware will not send any sample to a locator if the sample is filtered out by all the DataReaders receiving samples on that locator. However, if there is one DataReader to which the sample has to be sent, all the DataReaders on the locator will perform reader-side filtering for the incoming sample.

6. If you are using a custom filter (not the default one), it must be registered in the DomainParticipant of the DataWriter and the DataReader.

7. The DataWriter is not configured to use batching.

When batching is enabled, content filtering is always done on the reader side. See 7.5.2 BATCH QosPolicy (DDS Extension) on page 374.

### 5.4.3 Creating ContentFilteredTopics

To create a ContentFilteredTopic that uses the default SQL filter, use the DomainParticipant’s **create_contentfilteredtopic()** operation:

```c
DDS_ContentFilteredTopic *create_contentfilteredtopic(
    const char * name,
    const DDS_Topic * related_topic,
    const char * filter_expression,
    const DDS_StringSeq & expression_parameters)
```

Or, to use a custom filter or the builtin STRINGMATCH filter (see 5.4.7 STRINGMATCH Filter Expression Notation on page 245), use the **create_contentfilteredtopic_with_filter()** variation:

```c
DDS_ContentFilteredTopic *create_contentfilteredtopic_with_filter(
    const char * name,
    DDS_Topic * related_topic,
    const char * filter_expression,
    const DDS_StringSeq & expression_parameters,
    const char * filter_name = DDS_SQLFILTER_NAME)
```

Where:
5.4.3 Creating ContentFilteredTopics

name
Name of the ContentFilteredTopic. Note that it is legal for a ContentFilteredTopic to have the same name as a Topic in the same DomainParticipant, but a ContentFilteredTopic cannot have the same name as another ContentFilteredTopic in the same DomainParticipant. This parameter cannot be NULL.

related_topic
The related Topic to be filtered. The related topic must be in the same DomainParticipant as the ContentFilteredTopic. This parameter cannot be NULL. The same related topic can be used in many different ContentFilteredTopics.

filter_expression
A logical expression on the contents on the Topic. If the expression evaluates to TRUE, a DDS sample is received; otherwise it is discarded. This parameter cannot be NULL. The notation for this expression depends on the filter that you are using (specified by the filter_name parameter). See 5.4.6 SQL Filter Expression Notation on page 236 and 5.4.7 STRINGMATCH Filter Expression Notation on page 245. The filter_expression can be changed with set_expression() (5.4.5.2 Setting an Expression’s Filter and Parameters on page 234).

expression_parameters
A string sequence of filter expression parameters. Each parameter corresponds to a positional argument in the filter expression: element 0 corresponds to positional argument 0, element 1 to positional argument 1, and so forth.

The expression_parameters can be changed with set_expression_parameters() or set_expression() (5.4.5.2 Setting an Expression’s Filter and Parameters on page 234), append_to_expression_parameter() (5.4.5.3 Appending a String to an Expression Parameter on page 234) and remove_from_expression_parameter() (5.4.5.4 Removing a String from an Expression Parameter on page 235).

filter_name
Name of the content filter to use for filtering. The filter must have been previously registered with the DomainParticipant (see 5.4.10.2 Registering a Custom Filter on page 249). There are two builtin filters, DDS_SQLFILTER_NAME (the default filter) and DDS_STRINGMATCHFILTER_NAME—these are automatically registered.

To use the STRINGMATCH filter, call create_contentfilteredtopic_with_filter() with "DDS_STRINGMATCHFILTER_NAME" as the filter_name. STRINGMATCH filter expressions have the syntax:

\texttt{<field name> MATCH <string pattern>} (see 5.4.7 STRINGMATCH Filter Expression Notation on page 245).

To summarize:

- To use the builtin default SQL filter:
  - Call create_contentfilteredtopic()
  - See 5.4.6 SQL Filter Expression Notation on page 236

---

1 In the Java and C# APIs, you can access the names of the builtin filters by using DomainParticipant.SQLFILTER_NAME and DomainParticipant.STRINGMATCHFILTER_NAME.

2 In the Java and C# APIs, you can access the names of the builtin filters by using DomainParticipant.SQLFILTER_NAME and DomainParticipant.STRINGMATCHFILTER_NAME.
5.4.3 Creating ContentFilteredTopics

- To use the built-in STRINGMATCH filter:
  - Call `create_contentfilteredtopic_with_filter()`, setting the `filter_name` to `DDS_STRINGMATCHFILTER_NAME`
  - See 5.4.7 STRINGMATCH Filter Expression Notation on page 245

- To use a custom filter:
  - Call `create_contentfilteredtopic_with_filter()`, setting the `filter_name` to a registered custom filter

Be careful with memory management of the string sequence in some of the ContentFilteredTopic APIs. See the String Support section in the API Reference HTML documentation (within the Infrastructure module) for details on sequences.

5.4.3.1 Creating ContentFilteredTopics for Built-in DDS Types

To create a ContentFilteredTopic for a built-in DDS type (see 3.2 Built-in Data Types on page 38), use the standard `DomainParticipant` operations, `create_contentfilteredtopic()` or `create_contentfilteredtopic_with_filter()`.

The field names used in the filter expressions for the built-in SQL (see 5.4.6 SQL Filter Expression Notation on page 236) and StringMatch filters (see 5.4.7 STRINGMATCH Filter Expression Notation on page 245) must correspond to the names provided in the IDL description of the built-in DDS types.

ContentFilteredTopic Creation Examples:

For simplicity, error handling is not shown in the following examples.

C Example:

```c
DDS_Topic * topic = NULL;
DDS_ContentFilteredTopic * contentFilteredTopic = NULL;
struct DDS_StringSeq parameters = DDS_SEQUENCE_INITIALIZER;

/* Create a string ContentFilteredTopic */

topic = DDS_DomainParticipant_create_topic(
    participant, "StringTopic",
    DDS_StdStringSupport_get_type_name(),
    &DDS_TOPIC_QOS_DEFAULT,NULL,
    DDS_STATUS_MASK_NONE);

contentFilteredTopic =
    DDS_DomainParticipant_create_contentfilteredtopic(
        participant,
        "StringContentFilteredTopic",
        topic,
        "value = 'Hello World!'", &parameters);
```
### Traditional C++ Example with Namespaces:

```cpp
using namespace DDS;
...

/* Create a String ContentFilteredTopic */
Topic * topic = participant->create_topic(
    "StringTopic",
    StringTypeSupport::get_type_name(),
    TOPIC_QOS_DEFAULT,
    NULL, STATUS_MASK_NONE);
StringSeq parameters;
ContentFilteredTopic * contentFilteredTopic =
    participant->create_contentfilteredtopic(
    "StringContentFilteredTopic", topic,
    "value = 'Hello World!'", parameters);
```

### Modern C++ Example:

```cpp
using dds::core::StringTopicType;

dds::topic::Topic<StringTopicType> topic(participant, "StringTopic");
dds::topic::ContentFilteredTopic<StringTopicType> content_filtered_topic(
    topic,
    "StringContentFilteredTopic",
    dds::topic::Filter("value = 'Hello World!'"));
```

### C++/CLI Example:

```cpp
using namespace DDS;
...

/* Create a String ContentFilteredTopic */
Topic^ topic = participant->create_topic(
    "StringTopic",
    StringTypeSupport::get_type_name(),
    DomainParticipant::TOPIC_QOS_DEFAULT,
    nullptr, StatusMask::STATUS_MASK_NONE);
StringSeq^ parameters = gcnew StringSeq();
ContentFilteredTopic^ contentFilteredTopic =
    participant->create_contentfilteredtopic(
    "StringContentFilteredTopic", topic,
    "value = 'Hello World!'", parameters);
```

### C# Example:

```cpp
using namespace DDS;
...

/* Create a String ContentFilteredTopic */
Topic topic = participant.create_topic(
    "StringTopic",
    StringTypeSupport.get_type_name(),
    DomainParticipant.TOPIC_QOS_DEFAULT,
    null, StatusMask.STATUS_MASK_NONE);
StringSeq parameters = new StringSeq();
ContentFilteredTopic contentFilteredTopic =
    participant.create_contentfilteredtopic(
    "StringContentFilteredTopic", topic,
    "value = 'Hello World!'", parameters);
```

### Java Example:

```java
import com.rti.dds.type.builtin.*;
...
```
5.4.4 Deleting ContentFilteredTopics

To delete a ContentFilteredTopic, use the DomainParticipant’s `delete_contentfilteredtopic()` operation:

Make sure no DataReaders are using the ContentFilteredTopic. (If this is not true, the operation returns `PRECONDITION_NOT_MET`.)

Delete the ContentFilteredTopic by using the DomainParticipant’s `delete_contentfilteredtopic()` operation.

```c
DDS_ReturnCode_t delete_contentfilteredtopic
    (DDSContentFilteredTopic * a_contentfilteredtopic)
```

5.4.5 Using a ContentFilteredTopic

Once you’ve created a ContentFilteredTopic, you can use the operations listed in Table 5.5 ContentFilteredTopic Operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>append_to_expression_parameter</td>
<td>Concatenates a string value to the input expression parameter</td>
<td>5.4.5.3 Appending a String to an Expression Parameter on the next page</td>
</tr>
<tr>
<td>get_expression_parameters</td>
<td>Gets the expression parameters.</td>
<td>5.4.5.1 Getting the Current Expression Parameters on the next page</td>
</tr>
<tr>
<td>get_expression</td>
<td>Gets the expression.</td>
<td>5.4.5.5 Getting the Filter Expression on page 235</td>
</tr>
<tr>
<td>get_related_topic</td>
<td>Gets the related Topic.</td>
<td>5.4.5.6 Getting the Related Topic on page 235</td>
</tr>
<tr>
<td>narrow</td>
<td>Casts a DDS_TopicDescription pointer to a ContentFilteredTopic pointer.</td>
<td>5.4.5.7 ‘Narrowing’ a ContentFilteredTopic to a TopicDescription on page 235</td>
</tr>
<tr>
<td>remove_from_expression_parameter</td>
<td>Removes a string value from the input expression parameter</td>
<td>5.4.5.4 Removing a String from an Expression Parameter on page 235</td>
</tr>
<tr>
<td>set_expression</td>
<td>Changes the filter expression and parameters.</td>
<td>5.4.5.2 Setting an Expression’s Filter and Parameters on page 235</td>
</tr>
<tr>
<td>set_expression_parameters</td>
<td>Changes the expression parameters.</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.5 ContentFilteredTopic Operations
5.4.5 Using a ContentFilteredTopic

5.4.5.1 Getting the Current Expression Parameters

To get the expression parameters, use the ContentFilteredTopic’s `get_expression_parameters()` operation:

`DDS_ReturnCode_t get_expression_parameters(struct DDS_StringSeq & parameters)`

Where:

- `parameters` The filter expression parameters.

The memory for the strings in this sequence is managed as described in the `String Support` section of the API Reference HTML documentation (within the `Infrastructure` module). In particular, be careful to avoid a situation in which `Connext DDS` allocates a string on your behalf and you then reuse that string in such a way that `Connext DDS` believes it to have more memory allocated to it than it actually does. This parameter cannot be NULL.

This operation gives you the expression parameters that were specified on the last successful call to `set_expression_parameters()` or `set_expression()`, or if they were never called, the parameters specified when the ContentFilteredTopic was created.

5.4.5.2 Setting an Expression's Filter and Parameters

To change the filter expression and expression parameters associated with a ContentFilteredTopic:

`DDS_ReturnCode_t set_expression(
    const char * expression,
    const struct DDS_StringSeq & parameters)`

To change just the expression parameters (not the filter expression):

`DDS_ReturnCode_t set_expression_parameters(const struct DDS_StringSeq & parameters)`

Where:

- `expression` The new expression to be set in the ContentFilteredTopic.
- `parameters` The filter expression parameters. Each element in the parameter sequence corresponds to a positional parameter in the filter expression. When using the default `DDS_SQLFILTER_NAME`, parameter strings are automatically converted to the member type. For example, "4" is converted to the integer 4. This parameter cannot be NULL.

The ContentFilteredTopic’s operations do not manage the sequences; you must ensure that the parameter sequences are valid. Please refer to the `String Support` section in the API Reference HTML documentation (within the Infrastructure module) for details on sequences.

5.4.5.3 Appending a String to an Expression Parameter

To concatenate a string to an expression parameter, use the ContentFilteredTopic's `append_to_expression_parameter()` operation:
Using a ContentFilteredTopic

DDS_ReturnCode_t append_to_expression_parameter(const DDS_Long index, const char* value);

When using the STRINGMATCH filter, index must be 0.

This function is only intended to be used with the builtin SQL and STRINGMATCH filters. This function can be used in expression parameters associated with MATCH operators (see 5.4.6.5 SQL Extension: Regular Expression Matching on page 241) to add a pattern to the match pattern list. For example, if filter_expression is:

```c
symbol MATCH 'IBM'
```

Then append_to_expression_parameter(0, "MSFT") would generate the expression:

```c
symbol MATCH 'IBM,MSFT'
```

### 5.4.5.4 Removing a String from an Expression Parameter

To remove a string from an expression parameter use the ContentFilteredTopic's remove_from_expression_parameter() operation:

DDS_ReturnCode_t remove_from_expression_parameter(const DDS_Long index, const char* value)

When using the STRINGMATCH filter, index must be 0.

This function is only intended to be used with the builtin SQL and STRINGMATCH filters. It can be used in expression parameters associated with MATCH operators (see 5.4.6.5 SQL Extension: Regular Expression Matching on page 241) to remove a pattern from the match pattern list. For example, if filter_expression is:

```c
symbol MATCH 'IBM,MSFT'
```

Then remove_from_expression_parameter(0, "IBM") would generate the expression:

```c
symbol MATCH 'MSFT'
```

### 5.4.5.5 Getting the Filter Expression

To get the filter expression that was specified when the ContentFilteredTopic was created or when set_expression() was used:

```c
const char* get_filter_expression()
```

### 5.4.5.6 Getting the Related Topic

To get the related Topic that was specified when the ContentFilteredTopic was created:

```c
DDS_Topic * get_related_topic()
```

### 5.4.5.7 ‘Narrowing’ a ContentFilteredTopic to a TopicDescription

To safely cast a DDS_TopicDescription pointer to a ContentFilteredTopic pointer, use the ContentFilteredTopic’s narrow() operation:
5.4.6 SQL Filter Expression Notation

A SQL filter expression is similar to the WHERE clause in SQL. The SQL expression format provided by Connext DDS also supports the MATCH operator as an extended operator (see 5.4.6.5 SQL Extension: Regular Expression Matching on page 241).

The following sections provide more information:

- 5.4.6.1 Example SQL Filter Expressions below
- 5.4.6.2 SQL Grammar on page 238
- 5.4.6.3 Token Expressions on page 239
- 5.4.6.4 Type Compatibility in the Predicate on page 240
- 5.4.6.5 SQL Extension: Regular Expression Matching on page 241
- 5.4.6.6 Composite Members on page 242
- 5.4.6.7 Strings on page 243
- 5.4.6.8 Enumerations on page 243
- 5.4.6.9 Pointers on page 243
- 5.4.6.10 Arrays on page 243
- 5.4.6.12 Sequences on page 245

5.4.6.1 Example SQL Filter Expressions

Assume that you have a Topic with two floats, X and Y, which are the coordinates of an object moving inside a rectangle measuring 200 x 200 units. This object moves quite a bit, generating lots of DDS samples that you are not interested in. Instead you only want to receive DDS samples outside the middle of the rectangle, as seen in Figure 5.5: Filtering Example on the next page. That is, you want to filter out data points in the gray box.
The filter expression would look like this (remember the expression is written so that DDS samples that we do want will pass):

"(X < 50 or X > 150) and (Y < 50 or Y > 150)"

Suppose you would like the ability to adjust the coordinates that are considered outside the acceptable range (changing the size of the gray box). You can achieve this by changing the whole filter expression, using `set_expression()`, or by using filter parameters. The expression can be written using filter parameters as follows:

"(X < %0 or X > %1) and (Y < %2 or Y > %3)"

Recall that when you create a ContentFilteredTopic (see 5.4.3 Creating ContentFilteredTopics on page 229), you pass a `expression_parameters` string sequence as one of the parameters. Each element in the string sequence corresponds to one argument.

See the **String** and **Sequence Support** sections of the API Reference HTML documentation (from the **Modules** page, select **RTI Connext DDS API Reference, Infrastructure Module**).

In C++, the filter parameters could be assigned like this:

```c++
FilterParameter[0] = "50";
FilterParameter[1] = "150";
FilterParameter[2] = "50";
FilterParameter[3] = "150";
```
With these parameters, the filter expression is identical to the first approach. However, it is now possible to change the parameters by calling `set_expression_parameters()`. For example, perhaps you decide that you only want to see data points where \( X < 10 \) or \( X > 190 \). To make this change:

```plaintext
FilterParameter[0] = 10
FilterParameter[1] = 190
set_expression_parameters(...)
```

The new filter parameters will affect all `DataReaders` that have been created with this `ContentFilteredTopic`.

### 5.4.6.2 SQL Grammar

This section describes the subset of SQL syntax, in Backus–Naur Form (BNF), that you can use to form filter expressions.

The following notational conventions are used:

* **NonTerminals** are typeset in italics.

* 'Terminals' are quoted and typeset in a fixed-width font. They are written in upper case in most cases in the BNF-grammar below, but should be case insensitive.

* **TOKENS** are typeset in bold.

The notation (element `/ `,`) represents a non-empty, comma-separated list of elements.

```
FilterExpression ::= Condition
Condition ::= Predicate
     | Condition 'AND' Condition
     | Condition 'OR' Condition
     | 'NOT' Condition
     | '(' Condition ')'[^1]
Predicate ::= ComparisonPredicate
     | BetweenPredicate
ComparisonPredicate ::= ComparisonTerm RelOp ComparisonTerm
ComparisonTerm ::= FieldIdentifier
     | Parameter
BetweenPredicate ::= FieldIdentifier 'BETWEEN' Range
     | FieldIdentifier 'NOT BETWEEN' Range
FieldIdentifier ::= FIELDNAME
     | IDENTIFIER
RelOp ::= '=' | '>' | '>=' | '<' | '<=' | '<>' | 'LIKE' | 'MATCH'
Range ::= Parameter 'AND' Parameter
Parameter ::= INTEGERVALUE
     | CHARVALUE
     | FLOATVALUE
     | STRING
     | ENUMERATEDVALUE
     | BOOLEANVALUE
     | NULLVALUE
```

[^1]: For safety reasons, unbalanced parentheses are not accepted.
5.4.6 SQL Filter Expression Notation

5.4.6.3 Token Expressions

The syntax and meaning of the tokens used in SQL grammar is described as follows:

**IDENTIFIER**—An identifier for a FIELDNAME, defined as any series of characters 'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9', '_' but may not start with a digit.

**FIELDNAME**—A reference to a field in the data structure. A dot '.' is used to navigate through nested structures. The number of dots that may be used in a FIELDNAME is unlimited. The FIELDNAME can refer to fields at any depth in the data structure. The names of the field are those specified in the IDL definition of the corresponding structure, which may or may not match the fieldnames that appear on the language-specific (e.g., C/C++, Java) mapping of the structure. To reference the n+1 element in an array or sequence, use the notation ' [n] ', where n is a natural number (zero included). FIELDNAME must resolve to a primitive IDL type; that is boolean, octet, uint16, uint32, uint64, float double, char, wchar, string, wstring, or enum.

**TOPICNAME**—An identifier for a topic, and is defined as any series of characters 'a', ..., 'z', 'A', ..., 'Z', '0', ..., '9', '_' but may not start with a digit.

**INTEGERVALUE**—Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. 'L' or 'T' must be used for int64 (long long), otherwise

### Table: Predicate Data Type vs. IDL Type

<table>
<thead>
<tr>
<th>Predicate Data Type</th>
<th>IDL Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEANVALUE</td>
<td>boolean</td>
</tr>
<tr>
<td>INTEGERVALUE</td>
<td>octet, uint16, uint32, uint64</td>
</tr>
<tr>
<td>FLOATVALUE</td>
<td>float, double</td>
</tr>
<tr>
<td>CHARVALUE</td>
<td>char, wchar</td>
</tr>
<tr>
<td>STRING</td>
<td>string, wstring</td>
</tr>
<tr>
<td>ENUMERATEDVALUE</td>
<td>enum</td>
</tr>
</tbody>
</table>

Primitive IDL types referenced by FIELDNAME are treated as different types in Predicate according to the following table:
int32 (long) is assumed. A hexadecimal number is preceded by 0x and must be a valid hexadecimal expression.

INTEGERVALUE : ("+["+-",""])? (["0"-"9"])* (["L","l"])?
| ("+["+-",""])? ["0x","0X"](["0"-"9"],
"A"-"F", "a"-"f"])? (["L","l"])?

CHARVALUE—A single character enclosed between single quotes.

FLOATVALUE—Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point (‘.’). ‘F’ or ‘f’ must be used for float, otherwise double is assumed. A power-of-ten expression may be postfixed, which has the syntax en or En, where n is a number, optionally preceded by a plus or minus sign.

FLOATVALUE : ("+["+-",""])? (["0"-"9"])** (["]"). (["0"-"9"])* (EXPONENT)? (["e","E","f","F"])?

where EXPONENT: ["e","E"] (["+-",""])? (["0"-"9"])*

STRING—Any series of characters encapsulated in single quotes, except the single quote itself.

STRING : """ ("""")+ """

ENUMERATEDVALUE—A reference to a value declared within an enumeration. Enumerated values consist of the name of the enumeration label enclosed in single quotes. The name used for the enumeration label must correspond to the label names specified in the IDL definition of the enumeration.

ENUMERATEDVALUE : """ ["A"-"Z", "a"-"z"]
["A"-"Z", "a"-"z", _ _, "0"-"9"]* """

BOOLEANVALUE—Can either be TRUE or FALSE, and is case insensitive.

BOOLEANVALUE : ["TRUE","FALSE"]

NULLVALUE—Can be null, and is case insensitive.

NULLVALUE : "null"

PARAMETER—Takes the form %n, where n represents a natural number (zero included) smaller than 100. It refers to the (n + 1)th argument in the given context. This argument can only be in primitive type value format. It cannot be a FIELDNAME.

PARAMETER : "%n (["0"-"9"])*

5.4.6.4 Type Compatibility in the Predicate

As seen in Table 5.6 Valid Type Comparisons, only certain combinations of type comparisons are valid in the Predicate.
Table 5.6 Valid Type Comparisons

<table>
<thead>
<tr>
<th></th>
<th>BOOLEAN VALUE</th>
<th>INTEGER VALUE</th>
<th>FLOAT VALUE</th>
<th>CHAR VALUE</th>
<th>STRING</th>
<th>ENUMERATED VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOLEAN</td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER VALUE</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOAT VALUE</td>
<td></td>
<td>YES</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAR VALUE</td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td></td>
<td>YES</td>
</tr>
<tr>
<td>STRING</td>
<td></td>
<td></td>
<td></td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>ENUMERATED VALUE</td>
<td></td>
<td></td>
<td></td>
<td>YES²</td>
<td>YES³</td>
<td>YES⁴</td>
</tr>
</tbody>
</table>

5.4.6.5 SQL Extension: Regular Expression Matching

The relational operator MATCH may only be used with string fields. The right-hand operator is a string pattern. A string pattern specifies a template that the left-hand field must match.

MATCH is case-sensitive. The following characters have special meaning, unless escaped by the escape character: ', \, ?, *, [ ], - ^ ! \ %.

The pattern allows limited "wild card" matching under the rules in Table 5.7 Wild Card Matching.

The syntax is similar to the POSIX® fnmatch syntax. (See [http://www.open-group.org/onlinelibs/000095399/functions/fnmatch.html](http://www.open-group.org/onlinelibs/000095399/functions/fnmatch.html).) Some example expressions include:

This expression evaluates to TRUE if the value of symbol is equal to NASDAQ/MSFT:

```
symbol MATCH 'NASDAQ/MSFT'
```

This expression evaluates to TRUE if the value of symbol is equal to NASDAQ/IBM or NASDAQ/MSFT:

```
symbol MATCH 'NASD[Q]A[ ]MSF[T]'
```

---

³See 5.4.6.5 SQL Extension: Regular Expression Matching below.

²Because of the formal notation of the Enumeration values, they are compatible with string and char literals, but they are not compatible with string or char variables, i.e., "MyEnum=EnumValue" is correct, but "MyEnum=MyString" is not allowed.

³Because of the formal notation of the Enumeration values, they are compatible with string and char literals, but they are not compatible with string or char variables, i.e., "MyEnum=EnumValue" is correct, but "MyEnum=MyString" is not allowed.

⁴Only for same-type Enums.
symbol MATCH 'NASDAQ/IBM,NASDAQ/MSFT'

This expression evaluates to TRUE if the value of symbol is equal to NASDAQ and starts with a letter between M and Y:

symbol MATCH 'NASDAQ/[M-Y]*'

Table 5.7 Wild Card Matching

<table>
<thead>
<tr>
<th>Character</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>A , separates a list of alternate patterns. The field string is matched if it matches one or more of the patterns.</td>
</tr>
<tr>
<td>/</td>
<td>A / in the pattern string matches a / in the field string. It separates a sequence of mandatory substrings.</td>
</tr>
<tr>
<td>?</td>
<td>A ? in the pattern string matches any single non-special characters in the field string.</td>
</tr>
<tr>
<td>*</td>
<td>A * in the pattern string matches 0 or more non-special characters in field string.</td>
</tr>
<tr>
<td>%</td>
<td>This special character is used to designate filter expression parameters.</td>
</tr>
<tr>
<td>\</td>
<td>Escape character for special characters.</td>
</tr>
<tr>
<td>[charlist]</td>
<td>Matches any one of the characters in charlist.</td>
</tr>
<tr>
<td>[!charlist] or[^charlist]</td>
<td>(Not supported) Matches any one of the characters not in charlist.</td>
</tr>
<tr>
<td>[s-e]</td>
<td>Matches any character from s to e, inclusive.</td>
</tr>
<tr>
<td>[!s-e] or[^s-e]</td>
<td>(Not supported) Matches any character not in the interval s to e.</td>
</tr>
</tbody>
</table>

Note: To use special characters as regular characters in regular expressions, you must escape them using the character '\'. For example, 'A[A]' is considered a malformed expression and the result is undefined.

5.4.6.6 Composite Members

Any member can be used in the filter expression, with the following exceptions:

- 128-bit floating point numbers (long doubles) are not supported
- bitfields are not supported
- LIKE is not supported

Composite members are accessed using the familiar dot notation, such as "x.y.z > 5". For unions, the notation is special due to the nature of the IDL union type.

On the publishing side, you can access the union discriminator with myunion._d and the actual member with myunion._u.mymember. If you want to use a ContentFilteredTopic on the subscriber side and filter a DDS sample with a top-level union, you can access the union discriminator directly with _d and the actual member with mymember in the filter expression.
5.4.6 SQL Filter Expression Notation

5.4.6.7 Strings

The filter expression and parameters can use IDL strings. String constants must appear between single quotation marks (').

For example:

```
"fish = 'salmon'"
```

Strings used as parameter values must contain the enclosing quotation marks (') within the parameter value; do not place the quotation marks within the expression statement. For example, the expression " symbol MATCH %0 " with parameter 0 set to "IBM" is legal, whereas the expression " symbol MATCH '%0" with parameter 0 set to " IBM" will not compile.

5.4.6.8 Enumerations

A filter expression can use enumeration values, such as GREEN, instead of the numerical value. For example, if `x` is an enumeration of GREEN, YELLOW and RED, the following expressions are valid:

```
"x = 'GREEN'"
"X < 'RED'"
```

5.4.6.9 Pointers

Pointers can be used in filter expressions and are automatically dereferenced to the correct value.

For example:

```c
struct Point {
    int32 x;
    int32 y;
};
struct Rectangle {
    Point *u_l;
    Point *l_r;
};
```

The following expression is valid on a `Topic` of type Rectangle:

```
"u_l.x > l_r.x"
```

5.4.6.10 Arrays

Arrays are accessed with the familiar [] notation.

For example:

```c
struct ArrayType {
    int32 value[255][5];
};
```

The following expression is valid on a `Topic` of type ArrayType:
5.4.6 SQL Filter Expression Notation

"value[244][2] = 5"

In order to compare an array of bytes (octets in IDL), instead of comparing each individual element of the array using [] notation, Connext DDS provides a helper function, `hex()`. The `hex()` function can be used to represent an array of bytes (octets in IDL). To use the `hex()` function, use the notation `&hex()` and pass the byte array as a sequence of hexadecimal values.

For example:

```cpp
&hex (07 08 09 0A 0B 0c 0D 0E 10 11 12 13 14 15 16)
```

Here the leftmost-pair represents the byte at index 0.

**Note:** If the length of the octet array represented by the `hex()` function does not match the length of the field being compared, it will result in a compilation error.

For example:

```cpp
struct ArrayType {
    octet value[2];
};
```

The following expression is valid:

"value = &hex(12 0A)"

### 5.4.6.11 Optional Members

SQL filter expressions can refer to optional members. The syntax is the same as for any other member.

For example, given the type `MyType`:

```cpp
struct Foo {
    string text;
};
struct MyType {
    @optional int32 optional_member1;
    @optional Foo optional_member2;
    int32 non_optional_member;
};
```

These are valid expressions:

"optional_member1 = 1 AND optional_member2.text = 'hello' AND non_optional_member = 2"
"optional_member1 = null AND optional_member2.text <> null"

Any comparison involving an optional member (=, <>, <, or >) evaluates to false if the member is unset.

For example, both “optional_member1 <> 1” and “optional_member1 = 1” will evaluate to false if `optional_member1` is unset; however “optional_member1 = 1 OR non_optional_member = 1” will be true if `non_optional_member` is equal to 1 (even if `optional_member1` is unset). The expression “optional_member2.text = ‘hello’” will also be false if `optional_member2` is unset.
To check if an optional member is set or unset, you can compare with the null keyword. The following expressions are supported:

"optional_member1 = null" *, "optional_member1 <> null".

### 5.4.6.12 Sequences

Sequence elements can be accessed using the () or [] notation.

For example:

```cpp
struct SequenceType {
    sequence<int32> s;
};
```

The following expressions are valid on a `Topic` of type `SequenceType`:

"s[1] = 5"
"s[1] = 5"

### 5.4.7 STRINGMATCH Filter Expression Notation

The STRINGMATCH Filter is a subset of the SQL filter; it only supports the MATCH relational operator on a single string field. It is introduced mainly for the use case of partitioning data according to channels in the *DataWriter's 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension)* on page 432 in Market Data applications.

A STRINGMATCH filter expression has the following syntax:

`<field name> MATCH <string pattern>`

The STRINGMATCH filter is provided to support the narrow use case of filtering a single string field of the DDS sample against a comma-separated list of matching string values. It is intended to be used in conjunction with ContentFilteredTopic helper routines `append_to_expression_parameter()` (5.4.5.3 Appending a String to an Expression Parameter on page 234) and `remove_from_expression_parameter()` (5.4.5.4 Removing a String from an Expression Parameter on page 235), which allow you to easily append and remove individual string values from the comma-separated list of string values.

The STRINGMATCH filter must contain only one `<field name>`, and a single occurrence of the MATCH operator. The `<string pattern>` must be either the single parameter `%0`, or a single, comma-separated list of strings without intervening spaces.

During creation of a STRINGMATCH filter, the `<string pattern>` is automatically parameterized. That is, during creation, if the `<string pattern>` specified in the filter expression is not the parameter `%0`, then the comma-separated list of strings is copied to the initial contents of parameter 0 and the `<string pattern>` in the filter expression is replaced with the parameter `%0.`
The initial matching string list is converted to an explicit parameter value so that subsequent additions and deletions of string values to and from the list of matching strings may be performed with the `append_to_expression_parameter()` and `remove_from_expression_parameter()` operations mentioned above.

### 5.4.7.1 Example STRINGMATCH Filter Expressions

This expression evaluates to TRUE if the value of `symbol` is equal to `NASDAQ/MSFT`:

```plaintext
symbol MATCH 'NASDAQ/MSFT'
```

This expression evaluates to TRUE if the value of `symbol` is equal to `NASDAQ/IBM` or `NASDAQ/MSFT`:

```plaintext
symbol MATCH 'NASDAQ/IBM,NASDAQ/MSFT'
```

This expression evaluates to TRUE if the value of `symbol` is equal to `NASDAQ` and starts with a letter between M and Y:

```plaintext
symbol MATCH 'NASDAQ/[M-Y]*'
```

### 5.4.7.2 STRINGMATCH Filter Expression Parameters

In the builtin STRINGMATCH filter, there is one, and only one, parameter: parameter 0. (If you want to add more parameters, see 5.4.5.3 Appending a String to an Expression Parameter on page 234.) The parameter can be specified explicitly using the same syntax as the SQL filter or implicitly by using a constant string pattern. For example:

```plaintext
symbol MATCH %0 (Explicit parameter)
symbol MATCH 'IBM' (Implicit parameter initialized to IBM)
```

Strings used as parameter values must contain the enclosing quotation marks ('') within the parameter value; do not place the quotation marks within the expression statement. For example, the expression " symbol MATCH %0 " with parameter 0 set to " IBM " is legal, whereas the expression " symbol MATCH '%0' " with parameter 0 set to " IBM " will not compile.

### 5.4.8 Character Encoding

Connext DDS offers ISO 8859-1 as an alternative encoding for IDL strings. The default is UTF-8. To configure ISO 8859-1 for filtering of IDL strings, set the value of the `DomainParticipant's Property Qos property dds.domain_participant.filtering_character_encoding` to ISO-8859-1.

The possible values for `dds.domain_participant.filtering_character_encoding` are:

- UTF-8 (default value)
- ISO-8859-1
5.4.9 Unicode Normalization

This property is applicable to the following filtering features:

- ContentFilteredTopics (see 5.4 ContentFilteredTopics on page 227)
- Query conditions (see 4.6.7 ReadConditions and QueryConditions on page 209)
- TopicQueries (see Chapter 24 Topic Queries on page 977)
- MultiChannel DataWriters (see Chapter 20 Multi-channel DataWriters on page 896)

Unicode supports multiple ways to encode some characters, most notably accented characters. A composed character in Unicode can often have a number of different ways of representing the character. For example:

Precomposed Ĺ is represented by \u1e3c

Composed Ĺ = L + ^ is represented by \u004c + \u032d

The lexical comparison of the two characters above will return false. To do the correct comparison, the characters need to be normalized—that is, reduced to the same character composition.

When the character encoding for filtering of IDL strings is UTF-8, the Unicode normalization behavior can be controlled using a DomainParticipant Property Qos property called dds.domain_participant.filtering_unicode_normalization.

The possible values of the normalization property are:

- OFF: Disables normalization
- NFD: Canonical Decomposition
- NFC (default value): Canonical Decomposition, followed by Canonical Composition
- NFKC: Compatibility Decomposition, followed by Canonical Composition
- NFKC_Casefold: Casefold followed by NFKC normalization

This property is applicable to the following filtering features:

- ContentFilteredTopics (see 5.4 ContentFilteredTopics on page 227)
- Query conditions (see 4.6.7 ReadConditions and QueryConditions on page 209)
- TopicQueries (see Chapter 24 Topic Queries on page 977)
- MultiChannel DataWriters (see Chapter 20 Multi-channel DataWriters on page 896)
Because normalization may affect performance, and it is enabled by default, the property allows disabling the normalization process per DomainParticipant using the value OFF. However, be aware that doing this may lead to unexpected behavior.

### 5.4.10 Custom Content Filters

By default, a ContentFilteredTopic will use a SQL-like content filter, DDS_SQLFILTER_NAME (see 5.4.6 SQL Filter Expression Notation on page 236), which implements a superset of the content filter. There is another builtin filter, DDS_STRINGMATCHFILTER_NAME (see 5.4.7 STRINGMATCH Filter Expression Notation on page 245). Both of these are automatically registered.

If you want to use a different filter, **you must register it first**, then create the ContentFilteredTopic using `create_contentfilteredtopic_with_filter()` (see 5.4.3 Creating ContentFilteredTopics on page 229).

One reason to use a custom filter is that the default filter can only filter based on relational operations between topic members, not on a computation involving topic members. For example, if you want to filter based on the sum of the members, you must create your own filter.

**Note:**

- The API for using a custom content filter is subject to change in a future release.

#### 5.4.10.1 Filtering on the Writer Side with Custom Filters

There are two approaches for performing writer-side filtering. The first approach is to evaluate each written DDS sample against filters of all the readers that have content filter specified and identify the readers whose filter passes the DDS sample.

The second approach is to evaluate the written DDS sample once for the writer and then rely on the filter implementation to provide a set of readers whose filter passes the DDS sample. This approach allows the filter implementation to cache the result of filtering, if possible. For example, consider a scenario where the data is described by the struct shown below, where 10<x<20:

```c
struct MyData {
    int x;
    int y;
};
```

If the filter expression is based only on the x field, the filter implementation can maintain a hash map for all the different values of x and cache the filtering results in the hash map. Then any future evaluations will only be O(1), because it only requires a lookup in the hash map.

But if in the same example, a reader has a content filter that is based on both x and y, or just y, the filter implementation cannot cache the result—because the filter was only maintaining a hash map for x. In this case, the filter implementation can inform Connext DDS that it will not be caching the result for those DataReaders. The filter can use DDS_ExpressionProperty to indicate to the middleware whether or not it
will cache the results for DataReader. Table 5.8 DDS_ExpressionProperty describes DDS_ExpressionProperty.

<table>
<thead>
<tr>
<th>Table 5.8 DDS_ExpressionProperty</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>DDS_Boolean</td>
</tr>
<tr>
<td>DDS_Boolean</td>
</tr>
</tbody>
</table>

5.4.10.2 Registering a Custom Filter

To use a custom filter, it must be registered in the following places:

- Register the custom filter in any subscribing application in which the filter is used to create a ContentFilteredTopic and corresponding DataReader.
- In each publishing application, you only need to register the custom filter if you want to perform writer-side filtering. A DataWriter created with an associated filter will use that filter if it discovers a matched DataReader that uses the same filter.

For example, suppose Application A on the subscription side creates a Topic named X and a ContentFilteredTopic named filteredX (and a corresponding DataReader), using a previously registered content filter, myFilter. With only that, you will have filtering on the subscription side. If you also want to perform filtering in any application that publishes Topic X, then you also need to register the same definition of the ContentFilter myFilter in that application.

To register a new filter, use the DomainParticipant's register_contentfilter() operation:

```
DDS_ReturnCode_t register_contentfilter(
    const char * filter_name,
    const DDSContentFilter * contentfilter)
```

- **filter_name**

  The name of the filter. The name must be unique within the DomainParticipant. The filter_name cannot have a length of 0. The same filtering functions and handle can be registered under different names.

---

1This operation is an extension to the DDS standard.
5.4.10 Custom Content Filters

- **content_filter**

  This class specifies the functions that will be used to process the filter.

You must derive from the DDSContentFilter base class and implement the virtual compile below, evaluate below, and finalize below functions described below.

Optionally, you can derive from the DDSWriterContentFilter base class instead, to implement additional filtering operations that will be used by the DataWriter. When performing writer-side filtering, these operations allow a DDS sample to be evaluated once for the DataWriter, instead of evaluating the DDS sample for every DataReader that is matched with the DataWriter. An instance of the derived class is then used as an argument when calling register_contentfilter().

- **compile**

  The function that will be used to compile a filter expression and parameters. Connext DDS will call this function when a ContentFilteredTopic is created and when the filter parameters are changed. This parameter cannot be NULL. See 5.4.10.5 Compile Function on page 252. This is a member of DDSContentFilter and DDSWriterContentFilter.

- **evaluate**

  The function that will be called by Connext DDS each time a DDS sample is received. Its purpose is to evaluate the DDS sample based on the filter. This parameter cannot be NULL. See 5.4.10.6 Evaluate Function on page 253. This is a member of DDSContentFilter and DDSWriterContentFilter.

- **finalize**

  The function that will be called by Connext DDS when an instance of the custom content filter is no longer needed. This parameter may be NULL. See 5.4.10.7 Finalize Function on page 253. This is a member of DDSContentFilter and DDSWriterContentFilter.

- **writer_attach**

  The function that will be used to create some state required to perform filtering on the writer side using the operations provided in DDSWriterContentFilter. Connext DDS will call this function for every DataWriter; it will be called only the first time the DataWriter matches a DataReader using the specified filter. This function will not be called for any subsequent DataReaders that match the DataWriter and are using the same filter. See 5.4.10.8 Writer Attach Function on page 254. This is a member of DDSWriterContentFilter.

- **writer_detach**

  The function that will be used to delete any state created using the writer_attach function. Connext DDS will call this function when the DataWriter is deleted. See 5.4.10.9 Writer Detach Function on page 254. This is a member of DDSWriterContentFilter.
5.4.10 Custom Content Filters

- **writer_compile**

  The function that will be used by the `DataWriter` to compile filter expression and parameters provided by the reader. `Connext DDS` will call this function when the `DataWriter` discovers a `DataReader` with a `ContentFilteredTopic` or when a `DataWriter` is notified of a change in `DataReader`'s filter parameter. This function will receive as an input a `DDS_Cookie_t` which uniquely identifies the `DataReader` for which the function was invoked. See 5.4.10.10 Writer Compile Function on page 254. This is a member of DDSWriterContentFilter.

- **writer_evaluate**

  The function that will be called by `Connext DDS` every time a `DataWriter` writes a new DDS sample. Its purpose is to evaluate the DDS sample for all the readers for which the `DataWriter` is performing writer-side filtering and return the list of `DDS_Cookie_t` associated with the `DataReaders` whose filter pass the DDS sample. See 5.4.10.11 Writer Evaluate Function on page 255.

- **writer_return_loan**

  The function that will be called by `Connext DDS` to return the loan on a sequence of `DDS_Cookie_t` provided by the `writer_evaluate` function. See 5.4.10.12 Writer Return Loan Function on page 255. This is a member of DDSWriterContentFilter.

- **writer_finalize**

  The function that will be called by `Connext DDS` to notify the filter implementation that the `DataWriter` is no longer matching with a `DataReader` for which it was previously performing writer-side filtering. This will allow the filter to purge any state it was maintaining for the `DataReader`. See 5.4.10.13 Writer Finalize Function on page 256. This is a member of DDSWriterContentFilter.

5.4.10.3 Unregistering a Custom Filter

To unregister a filter, use the `DomainParticipant`'s `unregister_contentfilter()` operation\(^1\), which is useful if you want to reuse a particular filter name. (Note: You do not have to unregister the filter before deleting the parent `DomainParticipant`. If you do not need to reuse the filter name to register another filter, there is no reason to unregister the filter.)

```c
DDS_ReturnCode_t unregister_contentfilter(const char * filter_name)
```

| filter_name | The name of the previously registered filter. The name must be unique within the `DomainParticipant`. The `filter_name` cannot have a length of 0. |

If you attempt to unregister a filter that is still being used by a `ContentFilteredTopic`, `unregister_contentfilter()` will return `PRECONDITION_NOT_MET`.

\(^1\)This operation is an extension to the DDS standard.
If there are still existing discovered DataReaders with the same filter_name and the filter's compile function has previously been called on the discovered DataReaders, the filter’s finalize function will be called on those discovered DataReaders before the content filter is unregistered. This means filtering will be performed on the application that is creating the DataReader.

5.4.10.4 Retrieving a ContentFilter

If you know the name of a ContentFilter, you can get a pointer to its structure. If the ContentFilter has not already been registered, this operation will return NULL.

```c
DDS_ContentFilter *lookup_contentfilter (const char * filter_name)
```

5.4.10.5 Compile Function

The compile function specified in the ContentFilter will be used to compile a filter expression and parameters. Please note that the term ‘compile’ is intentionally defined very broadly. It is entirely up to you, as the user, to decide what this function should do. The only requirement is that the error_code parameter passed to the compile function must return OK on successful execution. For example:

```c
DDS_ReturnCode_t sample_compile_function(
    void ** new_compile_data, const char * expression,
    const DDS_StringSeq & parameters,
    const DDS_TypeCode * type_code,
    const char * type_class_name,
    void * old_compile_data)
{
    *new_compile_data = (void*)DDS_String_dup(parameters[0]);
    return DDS_RETCODE_OK;
}
```

Where:

- **new_compile_data**: A user-specified opaque pointer of this instance of the content filter. This value is passed to the evaluate and finalize functions.
- **expression**: An ASCII string with the filter expression the ContentFilteredTopic was created with. Note that the memory used by the parameter pointer is owned by Connext DDS. If you want to manipulate this string, you must make a copy of it first. Do not free the memory for this string.
- **parameters**: A string sequence of expression parameters used to create the ContentFilteredTopic. The string sequence is equal (but not identical) to the string sequence passed to create_contentfilteredtopic() (see expression_parameters in 5.4.3 Creating ContentFilteredTopics on page 229).
  
  The sequence passed to the compile function is owned by Connext DDS and must not be referred to outside the compile function.
- **type_code**: A pointer to the type code of the related Topic. A type code is a description of the topic members, such as their type (long, octet, etc.), but does not contain any information with respect to the memory layout of the structures. The type code can be used to write filters that can be used with any type. See 3.7 Using Generated Types without Connext DDS (Standalone) on page 152. [Note: If you are using the Java API, this parameter will always be NULL.]
5.4.10.6 Evaluate Function

The evaluate function specified in the ContentFilter will be called each time a DDS sample is received. This function’s purpose is to determine if a DDS sample should be filtered out (not put in the receive queue).

For example:

```c
DDS_Boolean sample_evaluate_function(
    void* compile_data,
    const void* sample,
    struct DDS_FilterSampleInfo * meta_data) {
    char *parameter = (char*)compile_data;
    DDS_Long x;
    Foo *foo_sample = (Foo*)sample;
    sscanf(parameter, "%d", &x);
    return (foo_sample->x > x ? DDS_BOOLEAN_FALSE : DDS_BOOLEAN_TRUE);
}
```

The function may use the following parameters:

- **compile_data** The last return value from the compile function for this instance of the content filter. Can be NULL.
- **sample** A pointer to a C structure with the data to filter. Note that the evaluate function always receives deserialized data.
- **meta_data** A pointer to the meta data associated with the DDS sample.

**Note:** Currently the meta_data field only supports related_sample_identity (described in Table 7.17 DDS_WriteParams_t).

5.4.10.7 Finalize Function

The finalize function specified in the ContentFilter will be called when an instance of the custom content filter is no longer needed. When this function is called, it is safe to free all resources used by this particular instance of the custom content filter.
For example:

```c
void sample_finalize_function ( void* compile_data) {
   /* free parameter string from compile function */
   DDS_String_free((char *)compile_data);
}
```

The `finalize` function may use the following optional parameters:

- `system_key` See 5.4.10.5 Compile Function on page 252.
- `handle` This is the opaque returned by the last call to the `compile` function.

### 5.4.10.8 Writer Attach Function

The `writer_attach` function specified in the WriterContentFilter will be used to create some state that can be used by the filter to perform writer-side filtering more efficiently. It is entirely up to you, as the implementer of the filter, to decide if the filter requires this state.

The function has the following parameter:

- `writer_filter_data` A user-specified opaque pointer to some state created on the writer side that will help perform writer-side filtering efficiently.

### 5.4.10.9 Writer Detach Function

The `writer_detach` function specified in the WriterContentFilter will be used to free up any state that was created using the `writer_attach` function.

The function has the following parameter:

- `writer_filter_data` A pointer to the state created using the `writer_attach` function.

### 5.4.10.10 Writer Compile Function

The `writer_compile` function specified in the WriterContentFilter will be used by a `DataWriter` to compile a filter expression and parameters associated with a `DataReader` for which the `DataWriter` is performing filtering. The function will receive as input a `DDS_Cookie_t` that uniquely identifies the `DataReader` for which the function was invoked.

The function has the following parameters:

- `writer_filter_data` A pointer to the state created using the `writer_attach` function.
- `prop` A pointer to `DDS_ExpressionProperty`. This is an output parameter. It allows you to indicate to Connext DDS if a filter expression can be optimized (as described in 5.4.10.1 Filtering on the Writer Side with Custom Filters on page 248).
5.4.10 Custom Content Filters

expression
An ASCII\textunderscore Z string with the filter expression the Content\textunderscore Filtered\textunderscore Topic was created with. Note that the memory used by the parameter pointer is owned by Connext DDS. If you want to manipulate this string, you must make a copy of it first. Do not free the memory for this string.

parameters
A string sequence of expression parameters used to create the Content\textunderscore Filtered\textunderscore Topic. The string sequence is equal (but not identical) to the string sequence passed to create\textunderscore content\textunderscore filtered\textunderscore topic\textunderscore () (see expression\textunderscore parameters in 5.4.3 Creating Content\textunderscore Filtered\textunderscore Topics on page 229). The sequence passed to the compile function is owned by Connext DDS and must not be referred to outside the writer\textunderscore compile function.

type\textunderscore code
A pointer to the type code of the related Topic. A type code is a description of the topic members, such as their type (long, octet, etc.), but does not contain any information with respect to the memory layout of the structures. The type code can be used to write filters that can be used with any type. See 3.7 Using Generated Types without Connext DDS (Standalone) on page 152. [Note: If you are using the Java API, this parameter will always be NULL.]

type\textunderscore class\textunderscore name
The fully qualified class name of the related Topic.

cookie
A DDS\textunderscore Cookie\textunderscore t to uniquely identify the Data\textunderscore Reader for which the writer\textunderscore compile function was called.

5.4.10.11 Writer Evaluate Function

The writer\textunderscore evaluate function specified in the Writer\textunderscore Content\textunderscore Filter will be used by a Data\textunderscore Writer to retrieve the list of Data\textunderscore Readers whose filter passed the DDS sample. The writer\textunderscore evaluate function returns a sequence of cookies which identifies the set of Data\textunderscore Readers whose filter passes the DDS sample.

The function has the following parameters:

writer\textunderscore filter\textunderscore data
A pointer to the state created using the writer\textunderscore attach function.

sample
A pointer to the data to be filtered. Note that the writer\textunderscore evaluate function always receives deserialized data.

meta\textunderscore data
A pointer to the meta\textunderscore data associated with the DDS sample.

Note: Currently the meta\textunderscore data field only supports related\textunderscore sample\textunderscore identity (described in Table 7.17 DDS\textunderscore Write\textunderscore Params\textunderscore t).

5.4.10.12 Writer Return Loan Function

Connext DDS uses the writer\textunderscore return\textunderscore loan function specified in the Writer\textunderscore Content\textunderscore Filter to indicate to the filter implementation that it has finished using the sequence of cookies returned by the filter\textquotesingle s writer\textunderscore evaluate function. Your filter implementation should not free the memory associated with the cookie sequence before the writer\textunderscore return\textunderscore loan function is called.

The function has the following parameters:

writer\textunderscore filter\textunderscore data
A pointer to the state created using the writer\textunderscore attach function.

cookies
The sequence of cookies for which the writer\textunderscore return\textunderscore loan function
was called.

5.4.10.13 Writer Finalize Function

The `writer_finalize` function specified in the WriterContentFilter will be called when the `DataWriter` no longer matches with a `DataReader` that was created with ContentFilteredTopic. This will allow the filter implementation to delete any state it was maintaining for the `DataReader`.

The function has the following parameters:

- **writer_filter_data**: A pointer to the state created using the `writer_attach` function.
- **cookie**: A `DDS_Cookie_t` to uniquely identify the `DataReader` for which the `writer_finalize` was called.
Chapter 6 Working with Instances

Instances are a way for an application to represent unique objects within a Topic, by specifying one or more key fields that form a unique identifier for the instance. Examples include identifying unique commercial flights within a “Flight Status” Topic or a unique sensor measuring the temperature in a “Temperature” Topic.

Modeling data using instances can provide several benefits to a system, including:

- An application can represent dynamic behavior of objects that come and go in a system, such as aircraft that may fly within range of a radar system and then fly out of range. See 6.1 Instance States on the next page for more details.
- Many QoS policies are applied per instance. For example, the 7.5.12 HISTORY QosPolicy on page 420 depth is applied per instance. This allows an application to specify: “Keep the last N samples for every instance this DataReader receives.” See 6.3.1 QoS Policies that are Applied per Instance on page 266 for more examples.
- An application can use DataReader methods such as read_instance() and take_instance() to process all the samples for an instance at once.
- ContentFilteredTopics are more efficient when filtering instances. Using ContentFilteredTopics in combination with instances is a great way to allow applications to take advantage of writer-side filtering to only subscribe to logical subsets of a Topic by specifying the instances that they are interested in.

Instances are defined by key fields that make up a unique identifier of the object being represented. Key fields are similar to primary keys in a database—each unique combination of key field values represents a unique instance. Key fields are specified using the @key directive, as shown in 2.4 DDS Samples, Instances, and Keys on page 18.
Table 6.1 Example Keys and Instances

<table>
<thead>
<tr>
<th>Instance (object represented in data)</th>
<th>Key (field(s) uniquely identifying object)</th>
<th>Data type</th>
<th>Sample (update to object)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commercial flight being tracked</td>
<td>Airline name and flight number, such as:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Airline: &quot;United Airlines&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Flight number: 901</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@key string airline</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@key int16 flight_num</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>float lat</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>float long</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UA, 901, 37.7749, 122.4194</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>UA, 901, 37.7748, 122.4195</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensor that is sending data, such as</td>
<td>Unique identifier of that sensor, such as:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>an individual temperature sensor</td>
<td>&quot;Floor-08-South&quot;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@key string sensor_id</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>int32 temperature</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floor-08-South, 78</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Floor-08-South, 79</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Car being monitored</td>
<td>Vehicle identification number (VIN) of the</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>car</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>@key string VIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>float lat</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>float long</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JH4DA9370MS016526, 37.7749, 122.4194</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JH4DA9370MS016526, 37.7748, 122.4195</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See the following sections:

- 6.1 Instance States below
- 6.2 Instance Memory Management on page 264
- 6.3 QoS Configuration and Instances on page 266

See also more details on instances from the DataWriter and DataReader perspectives:

- For the DataWriter: 7.3.14 Managing Instances (Working with Keyed Data Types) on page 325
- For the DataReader: 8.3.8 Accessing and Managing Instances (Working with Keyed Data Types) on page 534

6.1 Instance States

Instances can be in one of three states:

- **ALIVE**: An existing DataWriter has written a sample of the instance.
- **NOT_ALIVE_DISPOSED**: A DataWriter that has written the instance has called dispose() on the instance. (See 7.3.14.3 Disposing Instances on page 328 for further clarification when using EXCLUSIVE Ownership.)
- **NOT_ALIVE_NO_WRITERS**: All DataWriters that have written the instances have gone away (more on that later), or called unregister_instance() to unregister themselves from the instance.
6.1.1 ALIVE Details

The ALIVE instance state indicates that there is a *DataWriter* actively updating that instance, and no *DataWriter* has declared the instance to be “disposed” (see below).

An instance becomes ALIVE when a *DataWriter* writes a sample of that instance. This is true regardless of the previous state of the instance. For example, if an instance is NOT_ALIVE_DISPOSED, it becomes alive again when a *DataWriter* writes the instance. The only way for the instance to transition to becoming ALIVE is for a *DataWriter* to write a sample of that instance.

**Instances and OWNERSHIP QoS:** If the *DataWriters’ QoS* is set to OWNERSHIP = EXCLUSIVE, the *DataWriter* with the highest OWNERSHIP_STRENGTH that has written the instance is the owner of the instance, unless it unregisters the instance, loses liveness, or is deleted. If the instance has been disposed, only the *DataWriter* that owns that instance can make it transition to the ALIVE state by writing that instance. See 7.5.17 OWNERSHIP QosPolicy on page 434.

6.1.2 NOT_ALIVE_DISPOSED Details

The NOT_ALIVE_DISPOSED instance state indicates that a *DataWriter* has explicitly changed the state of an instance to NOT_ALIVE_DISPOSED by calling the `dispose()` method on the instance. The meaning of an instance becoming NOT_ALIVE_DISPOSED is part of the design of a system.
When a *DataWriter* calls `dispose()` on an instance, a dispose message is propagated from the *DataWriter* to its matching *DataReaders* to tell those *DataReaders* that the instance’s state is changed to **NOT_ALIVE_DISPOSED**.

Many systems use the **NOT_ALIVE_DISPOSED** instance state to indicate that the object that the instance represents has gone away. For example, in a “FlightData” topic, a system may use the **NOT_ALIVE_DISPOSED** instance state to indicate that the aircraft tracked by a radar system has flown out of range or has landed.

One common misconception is that the memory belonging to a disposed instance is immediately freed when the *DataWriter* calls `dispose()`. This is not true, because the dispose message needs to be propagated to *DataReaders*. This means that information about the instance—and the fact that it was disposed—is kept in the *DataWriter’s* queue based on QoS policies such as 7.5.21 RELIABILITY QosPolicy on page 447, 7.5.9 DURABILITY QosPolicy on page 411, and 7.5.12 HISTORY QosPolicy on page 420. See 6.3.2 QoS Policies that Affect Instance Management on page 267 for more information on managing resources for instances.

An instance can transition from **NOT_ALIVE_DISPOSED** to **ALIVE** if a *DataWriter* writes a new sample of that instance. An example of a system that transitions an instance to **NOT_ALIVE_DISPOSED** and then back to **ALIVE** is a radar system at an airport. It could be tracking a flight with the following key fields:

- airline = UA
- flight_num = 901

In this example, when the flight arrives on radar, the instance becomes **ALIVE**. When the flight lands, it becomes **NOT_ALIVE_DISPOSED**. The same flight flies every day, so it transitions from **NOT_ALIVE_DISPOSED** to **ALIVE** when the flight arrives again the next day. This maps to the state diagram shown in Figure 6.2: Instance State Diagram: Example for Flight Data on the next page.
Figure 6.2: Instance State Diagram: Example for Flight Data

**Instances and OWNERSHIP QoS:** If the DataWriter’s QoS policy is set to OWNERSHIP = EXCLUSIVE, the DataWriter with the highest OWNERSHIP_STRENGTH that has written the instance is the owner of the instance. It is also the only DataWriter that can dispose the instance. It does not lose ownership by disposing. Other DataWriters can call dispose(), but their dispose will have no effect on the instance state. OWNERSHIP is generally used for redundancy purposes, so it makes sense for only one owning DataWriter at a time to affect the instance state. See 7.5.17 OWNERSHIP QosPolicy on page 434 for further details.

### 6.1.3 NOT_ALIVE_NO_WRITERS Details

The NOT_ALIVE_NO_WRITERS instance state indicates that there are no active DataWriters that are currently updating the instance.
An instance becomes **NOT_ALIVE_NO_WRITERS** if all *DataWriters* that have written that instance have unregistered themselves from the instance or become not alive themselves (through losing liveliness, losing discovery liveliness, or being deleted). This means that if all *DataWriters* that have written samples for an instance are deleted, the instance changes state to **NOT_ALIVE_NO_WRITERS**.

Currently the state transition from **NOT_ALIVE_NO_WRITERS** to **ALIVE** happens only if new data is received, not if a previously-known writer is determined to be alive. Take for example a system where there is only a single *DataWriter* of an instance. If that *DataWriter* loses liveliness due to a temporary network disconnection, the *DataReaders* will detect that the instance is **NOT_ALIVE_NO_WRITERS**. When the network disconnection is resolved, the *DataReaders* will detect that the *DataWriter* has regained liveliness, but will not change the instance state to **ALIVE** until the *DataWriter* sends a new sample of that instance.

### 6.1.4 Transitions between NOT_ALIVE States

By default, there is no state transition between the **NOT_ALIVE_NO_WRITERS** and **NOT_ALIVE_DISPOSED** states, but this can be overridden by using the QoS settings `propagate_dispose_of_unregistered_instances` and `propagate_unregister_of_disposed_instances` on a *DataReader* via the 8.6.1 `DATA_READER_PROTOCOL` QosPolicy (DDS Extension) on page 570.

Setting `propagate_dispose_of_unregistered_instances` to true means that if all *DataWriters* lose liveliness (so the instance becomes **NOT_ALIVE_NO_WRITERS**), and then a *DataWriter* calls `dispose()` on the instance, the *DataReader* will recognize that instance as **NOT_ALIVE_DISPOSED** once the *DataWriter* regains liveliness.

Setting `propagate_dispose_of_unregistered_instances` to true could also mean that the first message a *DataReader* receives about an instance is **NOT_ALIVE_DISPOSED**. In Figure 6.3: Instance State Transitions: `propagate_dispose_of_unregistered_instances = true` on the next page, there is a new initial state transition from a *DataReader* never having seen an instance to seeing it as **NOT_ALIVE_DISPOSED**. In this case, the *DataReader* recognizes that the instance went from never existing (as far as the *DataReader* is concerned) to **NOT_ALIVE_DISPOSED**.

It is recommended that if you set `propagate_dispose_of_unregistered_instances` to true, you also set `serialize_key_with_dispose` to true (see 7.5.5 `DATA_WRITER_PROTOCOL` QosPolicy (DDS Extension) on page 389). This QoS will allow the subscribing application to retrieve the key value of the instance through the **FooDataReader_get_key_value** API, even though a valid sample for that instance has not been received.
6.1.4 Transitions between NOT_ALIVE States

Figure 6.3: Instance State Transitions: \texttt{propagate\_dispose\_of\_unregistered\_instances} = true

![Diagram of instance state transitions]

- Transitions shown with dashed lines are only available when \texttt{propagate\_dispose\_of\_unregistered\_instances} = true.

The \texttt{propagate\_unregister\_of\_disposed\_instances} QoS setting in the 8.6.1 \texttt{DATA\_READER\_PROTOCOL QosPolicy} (DDS Extension) on page 570 allows instances to transition directly from the instance being disposed to NOT_ALIVE_NO_WRITERS. See Figure 6.4: Instance State Transitions: \texttt{propagate\_unregister\_of\_disposed\_instances} = true on the next page. By default, only the resources for instances in the NOT_ALIVE_NO_WRITERS instance state are reclaimable in the DataReader queue.

In a system with finite instance resource limits, the \texttt{propagate\_unregister\_of\_disposed\_instances} setting allows an application to dispose instances to signal that the instance has gone away and then unregister those instances to make sure that the instances’ resources are reclaimable for use by new instances. Depending on your system requirements, another approach to reclaiming instance resources in the DataReader queue is to set \texttt{autopurge\_disposed\_instances\_delay} to zero. See 8.3.8.6 Instance Resource Limits and Memory Management on page 540 for more details.
6.2 Instance Memory Management

When an application creates keyed DataWriters and DataReaders (these are DataWriters and DataReaders whose Topics are keyed), Connext DDS needs to allocate memory for instance metadata. (Such metadata is not required for non-keyed data.) This includes memory for instance-specific metadata such as maintaining the current state of each instance and memory for instance keyhashes. Keyhashes are 16-byte representations of unique instances that are sent along with a sample. They allow DataWriters and DataReaders to quickly identify each unique instance without comparing all individual key fields.

The memory used for instances is separate from sample memory. A DataWriter or DataReader may have metadata stored for an instance even if there are currently no samples in the DataWriter's or DataReader's queue. Furthermore, memory related to instances and instance metadata is not deleted, but reclaimed. How memory is reclaimed for instances depends on how your QoS is set, and those QoS settings differ between DataWriters and DataReaders. For example, disposing an instance does not necessarily free up memory, depending on how your QoS is configured. (By default, the QoS settings do not free instance memory when instances are disposed.)

Transitions shown with dashed lines are only available when propagate_unregister_of_disposed_instances = true.
Consider a reliable, volatile `DataWriter` that writes a sample of an instance for the first time. The `DataWriter` stores the sample in its queue. At the same time, the `DataWriter` stores the keyhash for that instance, the state of the instance (ALIVE), and additional metadata about the instance. All matching `DataReader`S acknowledge the sample, so the `DataWriter` removes the sample from its queue, allowing that memory to be reused by another sample; however, the instance metadata is still valid, and continues to be stored.

Similarly, when each `DataReader` receives the first update about an instance, it stores the sample in its queue; it also stores the instance’s keyhash and state (ALIVE), and additional metadata about the instance. When the `DataReader` takes the sample from the queue, the sample is removed from the queue, allowing that memory to be reused by another sample; however, the instance metadata is still valid and continues to be stored.

Details on how `DataWriters` and `DataReaders` allocate and reclaim memory for instances can be found in the following sections:

- Details on the QoS policies that configure memory management on `DataWriters` are covered in 7.3.14.7 Instance Memory Management on page 330.
6.3 QoS Configuration and Instances

Some QoS policies are applied *per instance*, and other QoS policies configure instance management:

6.3.1 QoS Policies that are Applied per Instance

Several QoS policies (listed below) are applied per instance. This means that the QoS policy that’s specified on the `DataWriter` or `DataReader` is applied separately for each instance created. QoS policies cannot be specified *uniquely* per instance, however. For example, if you are representing airline flights as different instances, you can’t have a DEADLINE *period* of 1 second applied to one flight and a DEADLINE *period* of 2 seconds applied to another flight. The DEADLINE *period* (of, say, 1 second) is applied to each flight. In other words, you want to be notified if the flight position `DataReader` does not get an update about each individual flight within 1 second: the DEADLINE *period* is applied *per instance, for all instances*.

6.3.1.1 DEADLINE QosPolicy

The 7.5.7 DEADLINE QosPolicy on page 405 is checked separately for every instance. When notified of a missed deadline, a `DataWriter` or `DataReader` can check the last instance that missed the deadline using the instance handle in the status.

This allows a `DataWriter` to detect that it has not written a particular instance as frequently as it has offered in its deadline period, even if it has updated other instances.

This allows a `DataReader` to detect that it has not seen an update of an individual instance within the deadline period, even if it has seen updates from other instances during that time. This can be used to detect errors due to the `DataWriter` failing to write a particular instance. It can also detect network errors, where updates for a particular instance have been dropped or delayed.

6.3.1.2 DESTINATION_ORDER QosPolicy

The 7.5.8 DESTINATION_ORDER QosPolicy on page 408 contains a configuration option that allows a `DataWriter` or `DataReader` to order data across the whole Topic *for each instance*.

6.3.1.3 HISTORY QosPolicy

The 7.5.12 HISTORY QosPolicy on page 420 *depth* is applied for each instance created. For example, if *depth* = 1, the `DataWriter` or `DataReader` will keep one sample for each instance.

This allows an application to specify how much history it wants to keep per instance for reliability purposes. For example, if data is modeled as state data—meaning that only the most recent sample of the data
is important—the DataWriter and DataReader can set the history depth to 1. This allows them to send and receive only the most recent state for each instance.

### 6.3.1.4 DURABILITY QosPolicy

The 7.5.9 DURABILITY QosPolicy on page 411 writer_depth is applied for each instance created. For example, if writer_depth = 1, the DataWriter will keep one sample for each instance, for late-joining DataReaders.

This allows an application to specify how many samples it wants to keep per instance for later joiners. For example, if data is modeled as state data—meaning that only the most recent sample of the data is important—the DataWriter can set the writer_depth to 1. This allows it to send only the most recent state for each instance to late-joining DataReaders.

### 6.3.1.5 PRESENTATION QosPolicy

The 7.4.6 PRESENTATION QosPolicy on page 362 contains a configuration option to determine the scope of coherency and ordering of data in the DataReader’s queue. If coherency is enabled, this allows a Publisher or Subscriber to specify whether each coherent set is per instance. If ordered access is enabled, this allows a Publisher or Subscriber to specify whether data should be ordered per Topic or per instance.

### 6.3.1.6 TIME_BASED_FILTER QosPolicy

The 8.6.4 TIME_BASED_FILTER QosPolicy on page 588 filters out samples of each instance that arrive within the specified minimum_separation. For example, if the minimum_separation is 1 second, the DataReader will receive samples of instance A at most once a second, and samples of instance B at most once a second. A sample of instance A may arrive immediately after a sample of instance B, and will not be filtered out.

### 6.3.2 QoS Policies that Affect Instance Management

There are additional QoS policies that affect instances, primarily by controlling the limits or memory growth of instances, or by controlling which instance information is sent over the network.

#### 6.3.2.1 DataWriter and DataReader

The following policies affect both the DataWriter and DataReader.

##### 6.3.2.1.1 OWNERSHIP QosPolicy

If DataWriters have 7.5.17 OWNERSHIP QosPolicy on page 434 set to EXCLUSIVE, a DataWriter with higher OWNERSHIP_STRENGTH is the owner of any instances it writes. If a DataWriter calls unregister_instance(), it gives up ownership of the instance. If it calls dispose(), it does not give up ownership of the instance, so no other DataWriter can update that instance or its state.
6.3.2 QoS Policies that Affect Instance Management

6.3.2.1.2 RESOURCE_LIMITS QosPolicy

The 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 contains a field named max_instances that controls the maximum number of instances that may be stored for the DataWriter or DataReader.

6.3.2.2 DataWriter

The following policies apply to the DataWriter.

6.3.2.2.1 OWNERSHIP_STRENGTH QosPolicy

The DataWriter with highest 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438 will own the instances that it writes. This means that if a lower-strength DataWriter attempts to update any of those instances by writing or calling dispose on the instance, it does not affect the instance or its state.

6.3.2.2.2 DATA_WRITERRESOURCE_LIMITS QosPolicy

The instance_replacement and replace_empty_instances fields in the 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 401 control how instances can be replaced and the memory reclaimed if max_instances is reached. See 7.5.6.1 Configuring DataWriter Instance Replacement on page 404 for more information.

The autoregister_instances field controls whether to automatically register instances when a non-NIL handle is passed to the write() call.

6.3.2.2.3 WRITER_DATA_LIFECYCLE QosPolicy

The autodispose_unregistered_instances field in the 7.5.31 WRITER_DATA_LIFECYCLE QoS Policy on page 467 controls whether a DataWriter automatically disposes instances when they are unregistered. (By default, it doesn't.)

The autopurge_unregistered_instances_delay and autopurge_disposed_instances_delay fields control whether/when a DataWriter purges instances if they are NOT_ALIVE_NO_WRITERS or NOT_ALIVE_DISPOSED. Once all samples for an instance have been fully acknowledged by existing DataReaders, both the instance and the samples for that instance will be purged (see 7.3.8.2 write() behavior with KEEP_LAST and KEEP_ALL on page 312 for a definition of "fully ACK'ed").

See 7.3.14.7 Instance Memory Management on page 330 for more information on how this affects DataWriter memory usage.

6.3.2.2.4 DATA_WRITER_PROTOCOL QosPolicy

The disable_inline_keyhash field in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 controls whether or not a keyhash is propagated on the wire with each sample. This field allows the user to control whether bandwidth is used to send the keyhash with every sample, or CPU is used by the subscribing application to calculate the keyhash for every sample.
The `serialize_key_with_dispose` field controls whether or not the serialized key is propagated on the wire with dispose samples. This field is useful when `propagate_dispose_of_unregistered_instances` in the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570 is also true.

### 6.3.2.3 DataReader

The following policies apply to the `DataReader`.

#### 6.3.2.3.1 DATA_READER_PROTOCOL QosPolicy

The `propagate_unregister_of_disposed_instances` and `propagate_dispose_of_unregistered_instances` fields in the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570 control whether a `DataReader` can see state transitions between NOT_ALIVE instance states. See 6.1 Instance States on page 258 for more information.

#### 6.3.2.3.2 DATA_READERRESOURCE_LIMITS QosPolicy

The `initial_remote_writers_per_instance` and `max_remote_writers_per_instance` fields in the 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576 control the number of DataWriters from which a `DataReader` may receive samples for a single instance.

The `max_total_instances` field controls the maximum number of instances that a `DataReader` will maintain state for. See 8.3.8.6 Instance Resource Limits and Memory Management on page 540 for more information.

The `max_remote_virtual_writers_per_instance` field controls the maximum number of virtual remote writers that can be associated with an instance.

The `instance_replacement` field controls how instances can be replaced and the memory reclaimed if `max_instances` is reached. See 8.6.2.3 Configuring DataReader Instance Replacement on page 583 for more information.

#### 6.3.2.3.3 READER_DATA_LIFECYCLE QosPolicy

The `autopurge_nowriter_samples_delay` and `autopurge_disposed_samples_delay` fields in the 8.6.3 READER_DATA_LIFECYCLE QoS Policy on page 586 control whether/when to purge samples that are associated with instances in the `NOT_ALIVE_NO_WRITERS` or `NOT_ALIVE_DISPOSED` states, freeing up queue space and allowing instance information to be purged.

The `autopurge_disposed_instances_delay` field controls whether to purge instance memory when an instance becomes NOT_ALIVE_DISPOSED. The `autopurge_nowriter_instances_delay` field controls whether to purge instance memory when an instance becomes NOT_ALIVE_NOWRITERS.
Chapter 7 Sending Data

This section discusses how to create, configure, and use Publishers and DataWriters to send data. It describes how these Entities interact, as well as the types of operations that are available for them.

The goal of this section is to help you become familiar with the Entities you need for sending data. For up-to-date details such as formal parameters and return codes on any mentioned operations, please see the API Reference HTML documentation.

7.1 Preview: Steps to Sending Data

To send DDS samples of a data instance:

1. Create and configure the required Entities:
   a. Create a DomainParticipant (see 9.3.1 Creating a DomainParticipant on page 615).
   b. Register user data types\(^1\) with the DomainParticipant. For example, the ‘FooDataType’. (This step is not necessary in the Modern C++ API--the Topic instantiation automatically registers the type)
   c. Use the DomainParticipant to create a Topic with the registered data type.
   d. Optionally\(^2\), use the DomainParticipant to create a Publisher.
   e. Use the Publisher or DomainParticipant to create a DataWriter for the Topic.

\(^1\)Type registration is not required for built-in types (see 3.2.1 Registering Built-in Types on page 39).

\(^2\)You are not required to explicitly create a Publisher; instead, you can use the 'implicit Publisher' created from the DomainParticipant. See 7.2.1 Creating Publishers Explicitly vs. Implicitly on page 275.
f. Use a type-safe method to cast the generic `DataWriter` created by the `Publisher` to a type-specific `DataWriter`. For example, `FooDataWriter`. (This step doesn't apply to the Modern C++ API where you directly instantiate a type-safe `DataWriter<Foo>`.)

g. Optionally, register data instances with the `DataWriter`. If the `Topic`'s user data type contain `key` fields, then registering a data instance (data with a specific key value) will improve performance when repeatedly sending data with the same key. You may register many different data instances; each registration will return an instance handle corresponding to the specific key value. For non-keyed data types, instance registration has no effect. See 2.4 DDS Samples, Instances, and Keys on page 18 for more information on keyed data types and instances.

2. Every time there is changed data to be published:
   a. Store the data in a variable of the correct data type (for instance, variable `Foo` of the type `FooDataType`).
   b. Call the `FooDataWriter`'s `write()` operation, passing it a reference to the variable `Foo`.
      - For non-keyed data types or for non-registered instances, also pass in DDS_HANDLE_NIL.
      - For keyed data types, pass in the instance handle corresponding to the instance stored in `Foo`, if you have registered the instance previously. This means that the data stored in `Foo` has the same key value that was used to create instance handle.
   c. The `write()` function will take a snapshot of the contents of `Foo` and store it in Connext DDS internal buffers from where the DDS data sample is sent under the criteria set by the `Publisher`'s and `DataWriter`'s QosPolicies. If there are matched `DataReaders`, then the DDS data sample will have been passed to the physical transport plug-in/device driver by the time that `write()` returns.

### 7.2 Publishers

An application that intends to publish information needs the following Entities: DomainParticipant, Topic, Publisher, and DataWriter. All Entities have a corresponding specialized Listener and a set of QosPolicies. A Listener is how Connext DDS notifies your application of status changes relevant to the Entity. The QosPolicies allow your application to configure the behavior and resources of the Entity.

- A DomainParticipant defines the DDS domain in which the information will be made available.
- A Topic defines the name under which the data will be published, as well as the type (format) of the data itself.
- An application writes data using a DataWriter. The DataWriter is bound at creation time to a Topic, thus specifying the name under which the DataWriter will publish the data and the type associated
with the data. The application uses the `DataWriter’s write()` operation to indicate that a new value of the data is available for dissemination.

- A `Publisher` manages the activities of several `DataWriters`. The `Publisher` determines when the data is actually sent to other applications. Depending on the settings of various QoS Policies of the `Publisher` and `DataWriter`, data may be buffered to be sent with the data of other `DataWriters` or not sent at all. By default, the data is sent as soon as the `DataWriter’s write()` function is called.

You may have multiple `Publishers`, each managing a different set of `DataWriters`, or you may choose to use one `Publisher` for all your `DataWriters`.

For more information, see 7.2.1 Creating Publishers Explicitly vs. Implicitly on page 275.

**Figure 7.1:** Publication Module below shows how these `Entities` are related, as well as the methods defined for each `Entity`.

**Figure 7.1: Publication Module**

Publishers are used to perform the operations listed in Table 7.1 Publisher Operations on the next page. You can find more information about the operations by looking in the section listed under the Reference column. For details such as formal parameters and return codes, please see the API Reference HTML documentation.
Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

Table 7.1 Publisher Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataWriters</td>
<td>begin_coherent_changes</td>
<td>Indicates that the application will begin a coherent set of modifications.</td>
<td>7.3.10 Writing Coherent Sets of DDS Data Samples on page 316</td>
</tr>
<tr>
<td></td>
<td>create_datawriter</td>
<td>Creates a DataWriter that will belong to the Publisher.</td>
<td>7.3.1 Creating DataWriters on page 292</td>
</tr>
<tr>
<td></td>
<td>create_datawriter_with_profile</td>
<td>Sets the DataWriter's QoS based on a specified QoS profile.</td>
<td>7.2.4.6 Other Publisher QoS-Related Operations on page 283</td>
</tr>
<tr>
<td></td>
<td>copy_from_topic_qos</td>
<td>Copies relevant QosPolicies from a Topic into a DataWriterQos structure.</td>
<td>7.2.4.6 Other Publisher QoS-Related Operations on page 283</td>
</tr>
<tr>
<td>DataWriters</td>
<td>delete_contained_entities</td>
<td>Deletes all of the DataWriters that were created by the Publisher.</td>
<td>7.2.3.1 Deleting Contained DataWriters on page 278</td>
</tr>
<tr>
<td>cont'd</td>
<td>delete_datawriter</td>
<td>Deletes a DataWriter that belongs to the Publisher.</td>
<td>7.3.3 Deleting DataWriters on page 293</td>
</tr>
<tr>
<td></td>
<td>end_coherent_changes</td>
<td>Ends the coherent set initiated by begin_coherent_changes().</td>
<td>7.3.10 Writing Coherent Sets of DDS Data Samples on page 316</td>
</tr>
<tr>
<td></td>
<td>get_all_datawriters</td>
<td>Retrieves all the DataWriters created from this Publisher.</td>
<td>7.3.2 Getting All DataWriters on page 293</td>
</tr>
<tr>
<td></td>
<td>get_default_datawriter_qos</td>
<td>Copies the Publisher's default DataWriterQos values into a DataWriterQos structure.</td>
<td>7.3.15 Setting DataWriter QosPolicies on page 333</td>
</tr>
<tr>
<td></td>
<td>get_status_changes</td>
<td>Will always return 0 since there are no Statuses currently defined for Publishers.</td>
<td>4.1.4 Getting Status and Status Changes on page 171</td>
</tr>
<tr>
<td></td>
<td>lookup_datawriter</td>
<td>Retrieves a DataWriter previously created for a specific Topic.</td>
<td>7.2.6 Finding a Publisher's Related DDS Entities on page 286</td>
</tr>
</tbody>
</table>
## 7.2 Publishers

### Table 7.1 Publisher Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataWriters cont'd</td>
<td>set_default_datawriter_qos</td>
<td>Sets or changes the default DataWriterQos values.</td>
<td>7.2.4.5 Getting and Setting Default QoS for DataWriters on page 282</td>
</tr>
<tr>
<td></td>
<td>set_default_datawriter_qos_with_profile</td>
<td>Sets or changes the default DataWriterQos values based on a QoS profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wait_for_acknowledgments</td>
<td>Blocks until all data written by the Publisher’s reliable DataWriters are acknowledged by all matched reliable DataReaders, or until the a specified timeout duration, max_wait, elapses.</td>
<td>7.2.7 Waiting for Acknowledgments in a Publisher on page 286</td>
</tr>
<tr>
<td></td>
<td>is_sample_app_ acknowledged</td>
<td>Indicates if a sample has been application-acknowledged by all the matching DataReaders that were alive when the sample was written. If a DataReader does not enable application acknowledgment (by setting the ReliabilityQosPolicy's acknowledgment_kind to a value other than DDS_PROTOCOL_ACKNOWLEDGMENT_MODE), the sample is considered application-acknowledged for that DataReader.</td>
<td>7.3.12 Application Acknowledgment on page 317</td>
</tr>
<tr>
<td>Libraries and Profiles</td>
<td>get_default_library</td>
<td>Gets the Publisher’s default QoS profile library.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_profile</td>
<td>Gets the Publisher’s default QoS profile.</td>
<td>7.2.4.4 Getting and Setting the Publisher’s Default QoS Profile and Library on page 282</td>
</tr>
<tr>
<td></td>
<td>get_default_profile_library</td>
<td>Gets the library that contains the Publisher’s default QoS profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_library</td>
<td>Sets the default library for a Publisher.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_profile</td>
<td>Sets the default profile for a Publisher.</td>
<td></td>
</tr>
<tr>
<td>Participants</td>
<td>get_participant</td>
<td>Gets the DomainParticipant that was used to create the Publisher.</td>
<td>7.2.6 Finding a Publisher’s Related DDS Entities on page 286</td>
</tr>
</tbody>
</table>
Table 7.1 Publisher Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publishers</td>
<td>enable</td>
<td>Enables the Publisher.</td>
<td>4.1.2 Enabling DDS Entities on page 168</td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Compares two Publisher’s QoS structures for equality.</td>
<td>7.2.4.2 Comparing QoS Values on page 281</td>
</tr>
<tr>
<td></td>
<td>get_qos</td>
<td>Gets the Publisher’s current QoSPolicy settings. This is most often used in pre-preparation for calling set_qos().</td>
<td>7.2.4 Setting Publisher QosPolicies on page 278</td>
</tr>
<tr>
<td></td>
<td>set_qos</td>
<td>Sets the Publisher’s QoS. You can use this operation to change the values for the Publisher’s QoS Policies. Note, however, that not all QoS Policies can be changed after the Publisher has been created.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>Sets the Publisher’s QoS based on a specified QoS profile.</td>
<td></td>
</tr>
<tr>
<td>Publishers cont’d</td>
<td>get_listener</td>
<td>Gets the currently installed Listener.</td>
<td>7.2.5 Setting Up PublisherListeners on page 284</td>
</tr>
<tr>
<td></td>
<td>set_listener</td>
<td>Sets the Publisher’s Listener. If you created the Publisher without a Listener, you can use this operation to add one later.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>suspend_publications</td>
<td>Provides a hint that multiple data-objects within the Publisher are about to be written. Connext DDS does not currently use this hint.</td>
<td>7.2.9 Suspending and Resuming Publications on page 287</td>
</tr>
<tr>
<td></td>
<td>resume_publications</td>
<td>Reverses the action of suspend_publications().</td>
<td></td>
</tr>
</tbody>
</table>

7.2.1 Creating Publishers Explicitly vs. Implicitly

To send data, your application must have a Publisher. However, you are not required to explicitly create one. If you do not create one, the middleware will implicitly create a Publisher the first time you create a DataWriter using the DomainParticipant’s operations. It will be created with default QoS (DDS_PUBLISHER_QOS_DEFAULT) and no Listener.

A Publisher (implicit or explicit) gets its own default QoS and the default QoS for its child DataWriters from the DomainParticipant. These default QoS are set when the Publisher is created. (This is true for Subscribers and DataReaders, too.)

The 'implicit Publisher' can be accessed using the DomainParticipant’s get_implicit_publisher() operation (see 9.3.10 Getting the Implicit Publisher or Subscriber on page 629). You can use this ‘implicit Publisher’ just like any other Publisher (it has the same operations, QoS Policies, etc.). So you can change the mutable QoS and set a Listener if desired.

DataWriters are created by calling create_datawriter() or create_datawriter_with_profile()—these operations exist for DomainParticipants and Publishers. If you use the DomainParticipant to create a
DataWriter, it will belong to the implicit Publisher. If you use a Publisher to create a DataWriter, it will belong to that Publisher.

The middleware will use the same implicit Publisher for all DataWriters that are created using the DomainParticipant's operations.

Having the middleware implicitly create a Publisher allows you to skip the step of creating a Publisher. However, having all your DataWriters belong to the same Publisher can reduce the concurrency of the system because all the write operations will be serialized.

### 7.2.2 Creating Publishers

Before you can explicitly create a Publisher, you need a DomainParticipant (see 9.3 DomainParticipants on page 609). To create a Publisher, use the DomainParticipant's create_publisher() or create_publisher_with_profile() operations.

A QoS profile is a way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see Configuring QoS with XML (Chapter 19 on page 848).

Note: The Modern C++ API Publishers provide constructors whose first and only required argument is the DomainParticipant.

```c
DDSPublisher * create_publisher (  
    const DDS_PublisherQos &qos,  
    DDSPublisherListener *listener,  
    DDS_StatusMask mask)  

DDSPublisher * create_publisher_with_profile (  
    const char *library_name,  
    const char *profile_name,  
    DDSPublisherListener *listener,  
    DDS_StatusMask mask)
```

Where:

- **qos**
  - If you want the default QoS settings (described in the API Reference HTML documentation), use DDS_PUBLISHER_QOS_DEFAULT for this parameter (see Figure 7.2: Creating a Publisher with Default QosPolicies on the next page).
  - If you want to customize any of the QosPolicies, supply a QoS structure (see Figure 7.3: Creating a Publisher with Non-Default QosPolicies (not from a profile) on page 280). The QoS structure for a Publisher is described in 7.4 Publisher/Subscriber QosPolicies on page 345.
  - **Note:** If you use DDS_PUBLISHER_QOS_DEFAULT, it is not safe to create the Publisher while another thread may be simultaneously calling set_default_publisher_qos().

- **listener**
  - Listeners are callback routines. Connext DDS uses them to notify your application when specific events (status changes) occur with respect to the Publisher or the DataWriters created by the Publisher.
  - The listener parameter may be set to NULL if you do not want to install a Listener. If you use NULL, the Listener of the DomainParticipant to which the Publisher belongs will be used instead (if it is set). For more information on PublisherListeners, see 7.2.5 Setting Up PublisherListeners on page 284.
This bit-mask indicates which status changes will cause the Publisher's Listener to be invoked. The bits set in the mask must have corresponding callbacks implemented in the Listener.

If you use NULL for the Listener, use DDS_STATUS_MASK_NONE for this parameter. If the Listener implements all callbacks, use DDS_STATUS_MASK_ALL. For information on statuses, see 4.4 Listeners on page 189.

A QoS Library is a named set of QoS profiles. See 19.3 QoS Profiles on page 850. If NULL is used for library_name, the DomainParticipant's default library is assumed (see 7.2.4.4 Getting and Setting the Publisher's Default QoS Profile and Library on page 282).

A QoS profile groups a set of related QoS, usually one per entity. See 19.3 QoS Profiles on page 850. If NULL is used for profile_name, the DomainParticipant's default profile is assumed and library_name is ignored.

For more examples, see 7.2.4.1 Configuring QoS Settings when the Publisher is Created on page 279.

After you create a Publisher, the next step is to use the Publisher to create a DataWriter for each Topic, see 7.3.1 Creating DataWriters on page 292. For a list of operations you can perform with a Publisher, see Table 7.1 Publisher Operations.

7.2.3 Deleting Publishers

(Note: in the Modern C++ API, Entities are automatically destroyed, see 4.1.1 Creating and Deleting DDS Entities on page 167)

This section applies to both implicitly and explicitly created Publishers.

To delete a Publisher:

1. You must first delete all DataWriters that were created with the Publisher. Use the Publisher’s delete_datawriter() operation to delete them one at a time, or use the delete_contained_entities() operation (7.2.3.1 Deleting Contained DataWriters on the next page) to delete them all at the same time.

   DDS_ReturnCode_t delete_datawriter (DDSDataWriter *a_datawriter)

2. Delete the Publisher by using the DomainParticipant’s delete_publisher() operation.

   DDS_ReturnCode_t delete_publisher (DDSPublisher *p)
Note: A Publisher cannot be deleted within a Listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

7.2.3.1 Deleting Contained DataWriters

The Publisher’s delete_contained_entities() operation deletes all the DataWriters that were created by the Publisher.

```c
DDS_ReturnCode_t delete_contained_entities()
```

After this operation returns successfully, the application may delete the Publisher (see 7.2.3 Deleting Publishers on the previous page).

7.2.4 Setting Publisher QosPolicies

A Publisher’s QosPolicies control its behavior. Think of the policies as the configuration and behavior ‘properties’ of the Publisher. The DDS_PublisherQos structure has the following format:

```c
DDS_PublisherQos struct {
    DDS_PresentationQosPolicy presentation;
    DDS_PartitionQosPolicy partition;
    DDS_GroupDataQosPolicy group_data;
    DDS_EntityFactoryQosPolicy entity_factory;
    DDS_AsyncPublisherQosPolicy asynchronous_publisher;
    DDS_ExclusiveAreaQosPolicy exclusive_area;
    DDS_EntityNameQosPolicy publisher_name;
} DDS_PublisherQos;
```

Note: set_qos() cannot always be used in a listener callback; see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

Table 7.2 Publisher QosPolicies summarizes the meaning of each policy. (They appear alphabetically in the table.) For information on why you would want to change a particular QosPolicy, see the referenced section. For defaults and valid ranges, please refer to the API Reference HTML documentation for each policy.

Table 7.2 Publisher QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345</td>
<td>Configures the mechanism that sends user data in an external middleware thread.</td>
</tr>
<tr>
<td>7.4.2 ENTITYFACTORY QosPolicy on page 348</td>
<td>Controls whether or not child Entities are created in the enabled state.</td>
</tr>
<tr>
<td>7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418</td>
<td>Assigns a name and role_name to a Publisher.</td>
</tr>
</tbody>
</table>
### Table 7.2 Publisher QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) on page 350</td>
<td>Configures multi-thread concurrency and deadlock prevention capabilities.</td>
</tr>
<tr>
<td>7.4.4 GROUP_DATA QosPolicy on page 353</td>
<td>Along with 5.2.1 TOPIC_DATA QosPolicy on page 224 and 7.5.30 USER_DATA QosPolicy on page 465, this QosPolicy is used to attach a buffer of bytes to Connext DDS'a discovery meta-data.</td>
</tr>
<tr>
<td>7.4.5 PARTITION QosPolicy on page 356</td>
<td>Adds string identifiers that are used for matching DataReaders and DataWriters for the same Topic.</td>
</tr>
<tr>
<td>7.4.6 PRESENTATION QosPolicy on page 362</td>
<td>Controls how Connext DDS presents data received by an application to the DataReaders of the data.</td>
</tr>
</tbody>
</table>

#### 7.2.4.1 Configuring QoS Settings when the Publisher is Created

As described in 7.2.2 Creating Publishers on page 276, there are different ways to create a Publisher, depending on how you want to specify its QoS (with or without a QoS Profile).

- In Figure 7.2: Creating a Publisher with Default QosPolicies on page 277 we saw an example of how to explicitly create a Publisher with default QosPolicies. It used the special constant, DDS_PUBLISHER_QOS_DEFAULT, which indicates that the default QoS values for a Publisher should be used. Default Publisher QosPolicies are configured in the DomainParticipant; you can change them with the DomainParticipant’s set_default_publisher_qos() or set_default_publisher_qos_with_profile() operation (see 9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627).

- To create a Publisher with non-default QoS settings, without using a QoS profile, see Figure 7.3: Creating a Publisher with Non-Default QosPolicies (not from a profile) on the next page. It uses the DomainParticipant’s get_default_publisher_qos() method to initialize a DDS_PublisherQos structure. Then the policies are modified from their default values before the QoS structure is passed to create_publisher().

- You can also create a Publisher and specify its QoS settings via a QoS Profile. To do so, call create_publisher_with_profile(), as seen in Figure 7.4: Creating a Publisher with a QoS Profile on the next page.

- If you want to use a QoS profile, but then make some changes to the QoS before creating the Publisher, call the DomainParticipantFactory’s get_publisher_qos_from_profile(), modify the QoS and use the modified QoS structure when calling create_publisher(), as seen in Figure 7.5: Getting QoS Values from a Profile, Changing QoS Values, Creating a Publisher with Modified QoS Values on the next page.

For more information, see 7.2.2 Creating Publishers on page 276 and Configuring QoS with XML (Chapter 19 on page 848).
7.2.4 Setting Publisher QosPolicies

Figure 7.3: Creating a Publisher with Non-Default QosPolicies (not from a profile)

```c
DDS_PublisherQos publisher_qos;
// get defaults
if (participant->get_default_publisher_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
} // make QoS changes here
// for example, this changes the ENTITY_FACTORY QoS
publisher_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
    // create the publisher
    DDSPublisher* publisher = participant->create_publisher(publisher_qos,
        NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
}
```

Figure 7.4: Creating a Publisher with a QoS Profile

```c
// create the publisher with QoS profile
DDSPublisher* publisher = participant->create_publisher_with_profile(
    "MyPublisherLibrary", "MyPublisherProfile",
    NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
}
```

Figure 7.5: Getting QoS Values from a Profile, Changing QoS Values, Creating a Publisher with Modified QoS Values

```c
DDS_PublisherQos publisher_qos;
// Get publisher QoS from profile
retcode = factory->get_publisher_qos_from_profile(publisher_qos,
    "PublisherLibrary", "PublisherProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
} // Makes QoS changes here
// New entity_factory autoenable_created_entities will be true
publisher_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_TRUE;
    // create the publisher with modified QoS
    DDSPublisher* publisher = participant->create_publisher(
        "Example Foo", type_name, publisher_qos,
        NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
}
```

1 For the C API, you need to use DDS_PublisherQos_INITIALIZER or DDS_PublisherQos_initialize(). See 4.2.2 Special QosPolicy Handling Considerations for C on page 182

2 For the C API, you need to use DDS_PublisherQos_INITIALIZER or DDS_PublisherQos_initialize(). See 4.2.2 Special QosPolicy Handling Considerations for C on page 182
7.2.4 Setting Publisher QoS Policies

7.2.4.2 Comparing QoS Values

The `equals()` operation compares two `Publisher`’s DDS_PublisherQoS structures for equality. It takes two parameters for the two `Publisher`’s QoS structures to be compared, then returns TRUE if they are equal (all values are the same) or FALSE if they are not equal.

7.2.4.3 Changing QoS Settings After the Publisher Has Been Created

There are 2 ways to change an existing `Publisher`’s QoS after it is has been created—again depending on whether or not you are using a QoS Profile.

- To change an existing `Publisher`’s QoS programmatically (that is, without using a QoS profile): `get_qos()` and `set_qos()`. See the example code in Figure 7.6: Changing the QoS of an Existing Publisher below. It retrieves the current values by calling the `Publisher`’s `get_qos()` operation. Then it modify the value and call `set_qos()` to apply the new value. Note, however, that some QoS Policies cannot be changed after the `Publisher` has been enabled—this restriction is noted in the descriptions of the individual QoS Policies.

- You can also change a `Publisher`’s (and all other `Entities’`) QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see Figure 7.7: Changing the QoS of an Existing Publisher with a QoS Profile below. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

Figure 7.6: Changing the QoS of an Existing Publisher

```c
DDS_PublisherQos publisher_qos;
// Get current QoS. publisher points to an existing DDSPublisher.
if (publisher->get_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
// make changes
// New entity_factory autoenable_created_entities will be true
publisher_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_TRUE;
// Set the new QoS
if (publisher->set_qos(publisher_qos) != DDS_RETCODE_OK ) {
    // handle error
}
```

Figure 7.7: Changing the QoS of an Existing Publisher with a QoS Profile

```c
retcode = publisher->set_qos_with_profile(
    "PublisherProfileLibrary","PublisherProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

1 For the C API, you need to use DDS_PublisherQos_INITIALIZER or DDS_PublisherQos_initialize(). See 4.2.2 Special QoS Policy Handling Considerations for C on page 182
7.2.4 Setting Publisher QoS Policies

7.2.4.4 Getting and Setting the Publisher’s Default QoS Profile and Library

You can retrieve the default QoS profile used to create Publishers with the `get_default_profile()` operation.

You can also get the default library for Publishers, as well as the library that contains the Publisher’s default profile (these are not necessarily the same library); these operations are called `get_default_library()` and `get_default_library_profile()`, respectively. These operations are for informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see Configuring QoS with XML (Chapter 19 on page 848).

```cpp
virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_profile_library ()
```

There are also operations for setting the Publisher’s default library and profile:

```cpp
DDS_ReturnCode_t set_default_library (const char * library_name)
DDS_ReturnCode_t set_default_profile (const char * library_name,
const char * profile_name)
```

These operations only affect which library/profile will be used as the default the next time a default Publisher library/profile is needed during a call to one of this Publisher’s operations.

When calling a Publisher operation that requires a `profile_name` parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.) If the default library/profile is not set, the Publisher inherits the default from the DomainParticipant.

`set_default_profile()` does not set the default QoS for DataWriters created by the Publisher; for this functionality, use the Publisher’s `set_default_datawriter_qos_with_profile()`, see 7.2.4.5 Getting and Setting Default QoS for DataWriters below (you may pass in NULL after calling the Publisher’s `set_default_profile()`).

`set_default_profile()` does not set the default QoS for newly created Publishers; for this functionality, use the DomainParticipant’s `set_default_publisher_qos_with_profile()` operation, see 9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627.

7.2.4.5 Getting and Setting Default QoS for DataWriters

These operations set the default QoS that will be used for new DataWriters if `create_datawriter()` is called with DDS_DATAWRITER_QOS_DEFAULT as the `qos` parameter:

```cpp
DDS_ReturnCode_t set_default_datawriter_qos (const DDS_DataWriterQos &qos)
DDS_ReturnCode_t set_default_datawriter_qos_with_profile (
    const char *library_name,
    const char *profile_name)
```

The above operations may potentially allocate memory, depending on the sequences contained in some QoS policies.
To get the default QoS that will be used for creating DataWriter if `create_datawriter()` is called with DDS_PARTICIPANT_QOS_DEFAULT as the qos parameter:

```
DDS_ReturnCode_t get_default_datawriter_qos (DDS_DataWriterQos & qos)
```

This operation gets the QoS settings that were specified on the last successful call to `set_default_datawriter_qos()` or `set_default_datawriter_qos_with_profile()`, or if the call was never made, the default values listed in DDS_DataWriterQos.

**Note:** It is not safe to set the default DataWriter QoS values while another thread may be simultaneously calling `get_default_datawriter_qos()`, `set_default_datawriter_qos()`, or `create_datawriter()` with DDS_DATAWRITER_QOS_DEFAULT as the qos parameter. It is also not safe to get the default DataWriter QoS values while another thread may be simultaneously calling `set_default_datawriter_qos(0)`.

### 7.2.4.6 Other Publisher QoS-Related Operations

- **Copying a Topic’s QoS into a DataWriter’s QoS**

  This method is provided as a convenience for setting the values in a DataWriterQos structure before using that structure to create a DataWriter. As explained in 5.1.3 Setting Topic QosPolicies on page 219, most of the policies in a TopicQos structure do not apply directly to the Topic itself, but to the associated DataWriters and DataReaders of that Topic. The TopicQos serves as a single container where the values of QosPolicies that must be set compatibly across matching DataWriters and DataReaders can be stored.

  Thus instead of setting the values of the individual QosPolicies that make up a DataWriterQos structure every time you need to create a DataWriter for a Topic, you can use the Publisher’s copy_from_topic_qos() operation to “import” the Topic’s QosPolicies into a DataWriterQos structure. This operation copies the relevant policies in the TopicQos to the corresponding policies in the DataWriterQos.

  This copy operation will often be used in combination with the Publisher’s get_default_datawriter_qos() and the Topic’s get_qos() operations. The Topic’s QoS values are merged on top of the Publisher’s default DataWriter QosPolicies with the result used to create a new DataWriter, or to set the QoS of an existing one (see 7.3.15 Setting DataWriter QosPolicies on page 333).

- **Copying a Publisher’s QoS**

  C API users should use the DDS_PublisherQos_copy() operation rather than using structure assignment when copying between two QoS structures. The copy() operation will perform a deep copy so that policies that allocate heap memory such as sequences are copied correctly. In C++, C++/CLI, C# and Java, a copy constructor is provided to take care of sequences automatically.
• **Clearing QoS-Related Memory**

Some QoS Policies contain sequences that allocate memory dynamically as they grow or shrink. The C API’s DDS_PublisherQos_finalize() operation frees the memory used by sequences but otherwise leaves the QoS unchanged. C API users should call **finalize()** on all DDS_PublisherQos objects before they are freed, or for QoS structures allocated on the stack, before they go out of scope. In C++, C++/CLI, C# and Java, the memory used by sequences is freed in the destructor.

### 7.2.5 Setting Up PublisherListeners

Like all Entities, Publishers may optionally have **Listeners**. **Listeners** are user-defined objects that implement a DDS-defined interface (i.e. a pre-defined set of callback functions). **Listeners** provide the means for Connext DDS to notify applications of any changes in **Statuses** (events) that may be relevant to it. By writing the callback functions in the **Listener** and installing the **Listener** into the **Publisher**, applications can be notified to handle the events of interest. For more general information on **Listeners** and **Statuses**, see 4.4 **Listeners** on page 189.

**Note:** Some operations cannot be used within a listener callback, see 4.5.1 **Restricted Operations in Listener Callbacks** on page 200.

As illustrated in Figure 7.1: Publication Module on page 272, the **PublisherListener** interface extends the **DataWriterListener** interface. In other words, the **PublisherListener** interface contains all the functions in the **DataWriterListener** interface. There are no **Publisher**-specific **statuses**, and thus there are no **Publisher**-specific functions.

Instead, the methods of a **PublisherListener** will be called back for changes in the **Statuses** of any of the **DataWriters** that the **Publisher** has created. This is only true if the **DataWriter** itself does not have a **DataWriterListener** installed, see 7.3.4 **Setting Up DataWriterListeners** on page 294. If a **DataWriterListener** has been installed and has been enabled to handle a **Status** change for the **DataWriter**, then Connext DDS will call the method of the **DataWriterListener** instead.

If you want a **Publisher** to handle status events for its **DataWriters**, you can set up a **PublisherListener** during the **Publisher**’s creation or use the **set_listener()** method after the **Publisher** is created. The last parameter is a bit-mask with which you should set which **Status** events that the **PublisherListener** will handle. For example,

```c
DDS_StatusMask mask = DDS_OFFERED_DEADLINE_MISSED_STATUS | DDS_OFFERED_INCOMPATIBLE_QOS_STATUS;
publisher = participant->create_publisher(  
    DDS_PUBLISHER_QOS_DEFAULT, listener, mask);
```

or
7.2.5 Setting Up PublisherListeners

As previously mentioned, the callbacks in the PublisherListener act as ‘default’ callbacks for all the DataWriters contained within. When Connext DDS wants to notify a DataWriter of a relevant Status change (for example, PUBLICATION_MATCHED), it first checks to see if the DataWriter has the corresponding DataWriterListener callback enabled (such as the on_publication_matched() operation). If so, Connext DDS dispatches the event to the DataWriterListener callback. Otherwise, Connext DDS dispatches the event to the corresponding PublisherListener callback.

A particular callback in a DataWriter is not enabled if either:

- The application installed a NULL DataWriterListener (meaning there are no callbacks for the DataWriter at all).
- The application has disabled the callback for a DataWriterListener. This is done by turning off the associated status bit in the mask parameter passed to the set_listener() or create_datawriter() call when installing the DataWriterListener on the DataWriter. For more information on DataWriterListeners, see 7.3.4 Setting Up DataWriterListeners on page 294.

Similarly, the callbacks in the DomainParticipantListener act as ‘default’ callbacks for all the Publishers that belong to it. For more information on DomainParticipantListeners, see 9.3.6 Setting Up DomainParticipantListeners on page 620.

For example, Figure 7.8: Example Code to Create a Publisher with a Simple Listener below shows how to create a Publisher with a Listener that simply prints the events it receives.

**Figure 7.8: Example Code to Create a Publisher with a Simple Listener**

```cpp
class MyPublisherListener : public DDSPublisherListener {
public:
    virtual void on_offered_deadline_missed(
        DDSDataWriter* writer,
        const DDS_OfferedDeadlineMissedStatus& status);
    virtual void on_liveliness_lost(
        DDSDataWriter* writer,
        const DDS_LivelinessLostStatus& status);
    virtual void on_offered_incompatible_qos(
        DDSDataWriter* writer,
        const DDS_OfferedIncompatibleQosStatus& status);
    virtual void on_publication_matched(
        DDSDataWriter* writer,
        const DDS_PublicationMatchedStatus& status);
    virtual void on_reliable_writer_cache_changed(
        DDSDataWriter* writer,
        const DDS_ReliableWriterCacheChangedStatus& status);
    virtual void on_reliable_reader_activity_changed(
        DDSDataWriter* writer,
        const DDS_ReliableReaderActivityChangedStatus& status);
};
```
void MyPublisherListener::on_offered_deadline_missed(
    DDSDataWriter* writer,
    const DDS_OfferedDeadlineMissedStatus& status)
{
    printf("on_offered_deadline_missed
");
}
// ...Implement all remaining listeners in a similar manner...
DDSPublisherListener *myPubListener = new MyPublisherListener();
DDSPublisher* publisher =
    participant->create_publisher(DDS_PUBLISHER_QOS_DEFAULT,
                                 myPubListener, DDS_STATUS_MASK_ALL);

7.2.6 Finding a Publisher's Related DDS Entities

These Publisher operations are useful for obtaining a handle to related Entities:

- **get_participant()**: Gets the DomainParticipant with which a Publisher was created.
- **lookup_datawriter()**: Finds a DataWriter created by the Publisher with a Topic of a particular name. Note that in the event that multiple DataWriters were created by the same Publisher with the same Topic, any one of them may be returned by this method. (In the Modern C++ API this method is a freestanding function, 
  `dds::pub::find()`)
- **DDS_Publisher_as_Entity()**: This method is provided for C applications and is necessary when invoking the parent class Entity methods on Publishers. For example, to call the Entity method `get_status_changes()` on a Publisher, my_pub, do the following:

  ```cpp
  DDS_Entity get_status_changes(DDS_Publisher_as_Entity(my_pub))
  ```

`DDS_Publisher_as_Entity()` is not provided in the C++, C++/CLI, C# and Java APIs because the object-oriented features of those languages make it unnecessary.

7.2.7 Waiting for Acknowledgments in a Publisher

The Publisher's **wait_for_acknowledgments()** operation blocks the calling thread until either all data written by the Publisher's reliable DataWriters is acknowledged or the duration specified by the `max_wait` parameter elapses, whichever happens first.

Note that if a thread is blocked in the call to `wait_for_acknowledgments()` on a Publisher and a different thread writes new DDS samples on any of the Publisher’s reliable DataWriters, the new DDS samples must be acknowledged before unblocking the thread that is waiting on `wait_for_acknowledgments()`.

```cpp
DDS_ReturnCode_t wait_for_acknowledgments (const DDS_Duration_t & max_wait)
```

This operation returns `DDS_RETCODE_OK` if all the DDS samples were acknowledged, or `DDS_RETCODE_TIMEOUT` if the `max_wait` duration expired first.

There is a similar operation available for individual DataWriters, see 7.3.11 Waiting for Acknowledgments in a DataWriter on page 317.
The reliability protocol used by *Connext DDS* is discussed in *Reliable Communications* (Chapter 11 on page 688).

### 7.2.8 Statuses for Publishers

There are no statuses specific to the Publisher itself. The following statuses can be monitored by the *PublisherListener* for the Publisher’s *DataWriters*.

- 7.3.6.5 OFFERED_DEADLINE_MISSED Status on page 303
- 7.3.6.4 LIVELINESS_LOST Status on page 302
- 7.3.6.6 OFFERED_INCOMPATIBLE_QOS Status on page 303
- 7.3.6.7 PUBLICATION.MATCHED Status on page 304
- 7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305
- 7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) on page 307

### 7.2.9 Suspending and Resuming Publications

The operations *suspend_publications()* and *resume_publications()* provide a hint to *Connext DDS* that multiple data-objects within the Publisher are about to be written. *Connext DDS* does not currently use this hint.

### 7.3 DataWriters

To create a *DataWriter*, you need a *DomainParticipant* and a *Topic*.

You need a *DataWriter* for each *Topic* that you want to publish. Once you have a *DataWriter*, you can use it to perform the operations listed in Table 7.3 DataWriter Operations. The most important operation is *write()* , described in 7.3.8 Writing Data on page 309. For more details on all operations, see the API Reference HTML documentation.

*DataWriters* are created by using operations on a *DomainParticipant* or a *Publisher*, as described in 7.3.1 Creating DataWriters on page 292. If you use the *DomainParticipant’s* operations, the *DataWriter* will belong to an implicit *Publisher* that is automatically created by the middleware. If you use a *Publisher’s* operations, the *DataWriter* will belong to that *Publisher*. So either way, the *DataWriter* belongs to a *Publisher*.

**Note:** Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.
## Table 7.3 DataWriter Operations

<table>
<thead>
<tr>
<th>Working with</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataWriters</td>
<td>assert_liveliness</td>
<td>Manually asserts the liveliness of the DataWriter.</td>
<td>7.3.17 Asserting Liveliness on page 344</td>
</tr>
<tr>
<td></td>
<td>enable</td>
<td>Enables the DataWriter.</td>
<td>4.1.2 Enabling DDS Entities on page 168</td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Compares two DataWriter's QoS structures for equality.</td>
<td>7.3.15.2 Comparing QoS Values on page 338</td>
</tr>
<tr>
<td></td>
<td>get_qos</td>
<td>Gets the QoS.</td>
<td>7.3.15 Setting DataWriter QosPolicies on page 333</td>
</tr>
<tr>
<td></td>
<td>lookup_instance</td>
<td>Gets a handle, given an instance. (Useful for keyed data types only.)</td>
<td>7.3.14.5 Looking up an Instance Handle on page 330</td>
</tr>
<tr>
<td></td>
<td>set_qos</td>
<td>Modifies the QoS.</td>
<td>7.3.15 Setting DataWriter QosPolicies on page 333</td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>Modifies the QoS based on a QoS profile.</td>
<td>7.3.15 Setting DataWriter QosPolicies on page 333</td>
</tr>
<tr>
<td></td>
<td>get_listener</td>
<td>Gets the currently installed Listener.</td>
<td>7.3.4 Setting Up DataWriterListeners on page 294</td>
</tr>
<tr>
<td></td>
<td>set_listener</td>
<td>Replaces the Listener.</td>
<td></td>
</tr>
</tbody>
</table>
## Table 7.3 DataWriter Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>FooDataWriter (See 7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308)</td>
<td>dispose</td>
<td>States that the instance no longer exists. (Useful for keyed data types only.)</td>
<td>7.3.14.3 Disposing Instances on page 328</td>
</tr>
<tr>
<td></td>
<td>dispose_w_timestamp</td>
<td>Same as dispose, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>flush</td>
<td>Makes the batch available to be sent on the network.</td>
<td>7.3.9 Flushing Batches of DDS Data Samples on page 315</td>
</tr>
<tr>
<td></td>
<td>get_key_value</td>
<td>Maps an instance_handle to the corresponding key.</td>
<td>7.3.14.6 Getting the Key Value for an Instance on page 330</td>
</tr>
<tr>
<td></td>
<td>narrow</td>
<td>A type-safe way to cast a pointer. This takes a DDSDataWriter pointer and narrows it to a 'FooDataWriter' where 'Foo' is the related data type.</td>
<td>7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308</td>
</tr>
<tr>
<td></td>
<td>register_instance</td>
<td>States the intent of the DataWriter to write values of the data-instance that matches a specified key. Improves the performance of subsequent writes to the instance. (Useful for keyed data types only.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>register_instance_w_timestamp</td>
<td>Like register_instance, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)</td>
<td>7.3.14.2 Registering Instances on page 326 and 7.3.14.4 Unregistering Instances on page 328</td>
</tr>
<tr>
<td></td>
<td>unregister_instance</td>
<td>Reverses register_instance. Relinquishes the ownership of the instance. (Useful for keyed data types only.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unregister_instance_w_timestamp</td>
<td>Like unregister_instance, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>write</td>
<td>Writes a new value for a data-instance.</td>
<td>7.3.8 Writing Data on page 309</td>
</tr>
<tr>
<td></td>
<td>write_w_timestamp</td>
<td>Same as write, but allows the application to override the automatic source_timestamp.</td>
<td></td>
</tr>
<tr>
<td>FooDataWriter (See 7.3.7 Using a Type-Specific DataWriter (FooDataWriter) on page 308)</td>
<td>write_w_params</td>
<td>Same as write, but allows the application to specify parameters such as source_timestamp and instance_handle.</td>
<td>7.3.8 Writing Data on page 309</td>
</tr>
<tr>
<td></td>
<td>dispose_w_params</td>
<td>Same as dispose, but allows the application to specify parameters such as source_timestamp and instance_handle.</td>
<td>7.3.14.3 Disposing Instances on page 328</td>
</tr>
<tr>
<td></td>
<td>register_w_params</td>
<td>Same as register, but allows the application to specify parameters such as source_timestamp, instance_handle.</td>
<td>7.3.14.2 Registering Instances on page 326 and 7.3.14.4 Unregistering Instances on page 328</td>
</tr>
<tr>
<td></td>
<td>unregister_w_params</td>
<td>Same as unregister, but allows the application to specify parameters such as source_timestamp, and instance handle.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 7.3 DataWriter Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matched Subscriptions</td>
<td><code>get_matched_subscriptions</code></td>
<td>Gets a list of subscriptions that have a matching Topic and compatible QoS. These are the subscriptions currently associated with the DataWriter.</td>
<td>7.3.16.1 Finding Matching Subscriptions on page 342</td>
</tr>
<tr>
<td></td>
<td><code>get_matched_subscription_data</code></td>
<td>Gets information on a subscription with a matching Topic and compatible QoS.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>get_matched_subscription_locators</code></td>
<td>Gets a list of locators for subscriptions that have a matching Topic and compatible QoS. These are the subscriptions currently associated with the DataWriter.</td>
<td>7.3.16.2 Finding the Matching Subscription's Participant's BuiltInTopicData on page 343</td>
</tr>
<tr>
<td></td>
<td><code>get_matched_subscription_participant_data</code></td>
<td>Gets information about the DomainParticipant of a matching subscription.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>is_matched_subscription_active</code></td>
<td>Enables you to query whether the matched DataReader (identified using the instance handle returned by <code>get_matched_subscriptions</code>) is active. <code>get_matched_subscriptions</code> returns all matching DataReaders, including those that are not active. This operation enables you to see which matching DataReaders are active.</td>
<td>7.3.16.1 Finding Matching Subscriptions on page 342</td>
</tr>
<tr>
<td>Status</td>
<td><code>get_status_changes</code></td>
<td>Gets a list of statuses that have changed since the last time the application read the status or the listeners were called.</td>
<td>4.1.4 Getting Status and Status Changes on page 171</td>
</tr>
</tbody>
</table>
### Table 7.3 DataWriter Operations

<table>
<thead>
<tr>
<th>Working with</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>get_liveliness_lost_status</td>
<td>Gets LIVELINESS_LOST status.</td>
<td>7.3.6 Statuses for DataWriters on page 296</td>
</tr>
<tr>
<td>Status</td>
<td>get_offered_deadline_missed_status</td>
<td>Gets OFFERED_DEADLINE MISSED status.</td>
<td></td>
</tr>
<tr>
<td>cont’d</td>
<td>get_offered_incompatible_qos_status</td>
<td>Gets OFFERED_INCOMPATIBLE_QOS status.</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>get_publication_match_status</td>
<td>Gets PUBLICATION.MATCHED_QOS status.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_reliable_writer_cache_changed_status</td>
<td>Gets RELIABLE_WRITER_CACHE_CHANGED status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_reliable_reader_activity_changed_status</td>
<td>Gets RELIABLE_READER_ACTIVITY_CHANGED status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_datawriter_cache_status</td>
<td>Gets DATA_WRITER_CACHE status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_datawriter_protocol_status</td>
<td>Gets DATA_WRITER_PROTOCOL status</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_matched_subscription_datawriter_protocol_status</td>
<td>Gets DATA_WRITER_PROTOCOL status for this DataWriter, per matched subscription identified by the subscription_handle.</td>
<td>7.3.6 Statuses for DataWriters on page 296</td>
</tr>
<tr>
<td></td>
<td>get_matched_subscription_datawriter_protocol_status_by_locator</td>
<td>Gets DATA_WRITER_PROTOCOL status for this DataWriter, per matched subscription as identified by a locator.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_publisher</td>
<td>Gets the Publisher to which the DataWriter belongs.</td>
<td>7.3.16.3 Finding Related DDS Entities on page 344</td>
</tr>
<tr>
<td></td>
<td>get_topic</td>
<td>Get the Topic associated with the DataWriter.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>wait_for_acknowledgements</td>
<td>Blocks the calling thread until either all data written by the DataWriter is acknowledged by all matched Reliable DataReaders, or until the a specified timeout duration, max_wait, elapses.</td>
<td>7.3.11 Waiting for Acknowledgments in a DataWriter on page 317</td>
</tr>
</tbody>
</table>
7.3.1 Creating DataWriters

Before you can create a DataWriter, you need a DomainParticipant, a Topic, and optionally, a Publisher. DataWriters are created by calling `create_datawriter()` or `create_datawriter_with_profile()`—these operations exist for DomainParticipants and Publishers. If you use the DomainParticipant to create a DataWriter, it will belong to the implicit Publisher described in 7.2.1 Creating Publishers Explicitly vs. Implicitly on page 275. If you use a Publisher’s operations to create a DataWriter, it will belong to that Publisher.

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see Configuring QoS with XML (Chapter 19 on page 848).

Note: In the Modern C++ API DataWriters provide constructors whose first argument is a Publisher. The only required arguments are the publisher and the topic.

```
DDSDataWriter* create_datawriter (  
    DDSesion *topic,  
    const DDS_DataWriterQos &qos,  
    DDSDataWriterListener *listener,  
    DDS_StatusMask mask)

DDSDataWriter * create_datawriter_with_profile(  
    DDS_topic * topic,  
    const char * library_name,  
    const char * profile_name,  
    DDSDataWriterListener * listener,  
    DDS_StatusMask mask)
```

Where:

- **Topic** The Topic that the DataWriter will publish. This must have been previously created by the same DomainParticipant.

- **qos** If you want the default QoS settings (described in the API Reference HTML documentation), use the constant DDS_DATAWRITER_QOS_DEFAULT for this parameter (see Figure 7.9: Creating a DataWriter with Default QosPolicies and a Listener on the next page). If you want to customize any of the QoS policies, supply a QoS structure (see 7.3.15 Setting DataWriter QoS on page 333).

  **Note:** If you use DDS_DATAWRITER_QOS_DEFAULT for the qos parameter, it is not safe to create the DataWriter while another thread may be simultaneously calling the Publisher’s `set_default_datawriter_qos()` operation.

- **listener** Listeners are callback routines. Connext DDS uses them to notify your application of specific events (status changes) that may occur with respect to the DataWriter. The listener parameter may be set to NULL; in this case, the PublisherListener (or if that is NULL, the DomainParticipantListener) will be used instead. For more information, see 7.3.4 Setting Up DataWriterListeners on page 294.

- **mask** This bit-mask indicates which status changes will cause the Listener to be invoked. The bits set in the mask must have corresponding callbacks implemented in the Listener. If you use NULL for the Listener, use DDS_STATUS_MASK_NONE for this parameter. If the Listener implements all callbacks, use DDS_STATUS_MASK_ALL. For information on statuses, see 4.4 Listeners on page 189.

- **Library_name** A QoS Library is a named set of QoS profiles. See 19.3 QoS Profiles on page 850.
7.3.2 Getting All DataWriters

profile_name  A QoS profile groups a set of related QoS, usually one per entity. See 19.3 QoS Profiles on page 850.

For more examples on how to create a DataWriter, see 7.3.15.1 Configuring QoS Settings when the DataWriter is Created on page 337.

After you create a DataWriter, you can use it to write data. See 7.3.8 Writing Data on page 309.

Note: When a DataWriter is created, only those transports already registered are available to the DataWriter. The built-in transports are implicitly registered when (a) the DomainParticipant is enabled, (b) the first DataWriter is created, or (c) you look up a built-in data reader, whichever happens first.

Figure 7.9: Creating a DataWriter with Default QosPolicies and a Listener

```c++
// MyWriterListener is user defined, extends DDSDataWriterListener
DDSDataWriterListener* writer_listener = new MyWriterListener();
DDSDataWriter* writer = publisher->create_datawriter(
    topic,
    DDS_DATAWRITER_QOS_DEFAULT,
    writer_listener,
    DDS_STATUS_MASK_ALL);
if (writer == NULL) {
    // ... error
}
// narrow it for your specific data type
FooDataWriter* foo_writer = FooDataWriter::narrow(writer);
```

7.3.2 Getting All DataWriters

To retrieve all the DataWriters created by the Publisher, use the Publisher’s get_all_datawriters() operation:

```
DDS_ReturnCode_t get_all_datawriters(DDS_Publisher* self,
    struct DDS_DataWriterSeq* writers);
```

In the Modern C++ API, use the freestanding function rti::pub::find_datawriters().

7.3.3 Deleting DataWriters

(Note: in the Modern C++ API, Entities are automatically destroyed, see 4.1.1 Creating and Deleting DDS Entities on page 167)

To delete a single DataWriter, use the Publisher’s delete_datawriter() operation:

```
DDS_ReturnCode_t delete_datawriter (DDSDataWriter *a_datawriter)
```

Note: A DataWriter cannot be deleted within its own writer listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200

To delete all of a Publisher's DataWriters, use the Publisher's delete_contained_entities() operation (see 7.2.3.1 Deleting Contained DataWriters on page 278).
7.3.4 Setting Up DataWriterListeners

Data Writers may optionally have Listeners. Listeners are essentially callback routines and provide the means for Connext DDS to notify your application of the occurrence of events (status changes) relevant to the DataWriter. For more general information on Listeners, see 4.4 Listeners on page 189.

Note: Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

If you do not implement a DataWriterListener, the associated PublisherListener is used instead. If that Publisher also does not have a Listener, then the DomainParticipant’s Listener is used if one exists (see 7.2.5 Setting Up PublisherListeners on page 284 and 9.3.6 Setting Up DomainParticipantListeners on page 620).

Listeners are typically set up when the DataWriter is created (see 7.2 Publishers on page 271). You can also set one up after creation by using the set_listener() operation. Connext DDS will invoke a DataWriter’s Listener to report the status changes listed in Table 7.4 DataWriterListener Callbacks (if the Listener is set up to handle the particular status, see 7.3.4 Setting Up DataWriterListeners above).

Table 7.4 DataWriterListener Callbacks

<table>
<thead>
<tr>
<th>This DataWriterListener callback...</th>
<th>... is triggered by ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>on_instance_replaced()</td>
<td>A replacement of an existing instance by a new instance; see 7.5.6.1 Configuring DataWriter Instance Replacement on page 404</td>
</tr>
<tr>
<td>on_liveliness_lost</td>
<td>A change to 7.3.6.4 LIVELINESS_LOST Status on page 302</td>
</tr>
<tr>
<td>on_offered_deadline_missed</td>
<td>A change to 7.3.6.5 OFFERED_DEADLINE_MISSED Status on page 303</td>
</tr>
<tr>
<td>on_offered_incompatible_qos</td>
<td>A change to 7.3.6.6 OFFERED_INCOMPATIBLE_QOS Status on page 303</td>
</tr>
<tr>
<td>on_publication_matched</td>
<td>A change to 7.3.6.7 PUBLICATION_Mmatched Status on page 304</td>
</tr>
<tr>
<td>on_reliable_writer_cache_changed</td>
<td>A change to 7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305</td>
</tr>
<tr>
<td>on_reliable_reader_activity_changed</td>
<td>A change to 7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) on page 307</td>
</tr>
<tr>
<td>on_sample_removed</td>
<td>Removal of a sample from the DataWriter queue, when the sample was written with a cookie using the write_w_params API or if the DataWriter supports loaned samples, which are used by Zero Copy over shared memory and FlatData language binding (see Chapter 23 Sending Large Data on page 943)</td>
</tr>
<tr>
<td>on_application_acknowledgment</td>
<td>Application acknowledgment (see 7.3.6.1 APPLICATION_ACKNOWLEDGMENT_STATUS on page 296)</td>
</tr>
</tbody>
</table>
7.3.5 Checking DataWriter Status

You can access an individual communication status for a DataWriter with the operations shown in Table 7.5 DataWriter Status Operations.

Table 7.4 DataWriterListener Callbacks

<table>
<thead>
<tr>
<th>This DataWriterListener callback...</th>
<th>... is triggered by ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>on_service_request_accepted</td>
<td>A change to 7.3.6.10 SERVICE_REQUEST_ACCEPTED Status (DDS Extension) on page 308.</td>
</tr>
</tbody>
</table>

Table 7.5 DataWriter Status Operations

<table>
<thead>
<tr>
<th>Use this operation...</th>
<th>...to retrieve this status:</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_datawriter_cache_status</td>
<td>7.3.6.2 DATA_WRITER_CACHE_STATUS on page 297</td>
</tr>
<tr>
<td>get_datawriter_protocol_status</td>
<td>7.3.6.3 DATA_WRITER_PROTOCOL_STATUS on page 298</td>
</tr>
<tr>
<td>get_matched_subscription_datawriter_protocol_status</td>
<td>7.3.6.4 LIVELINESS_LOST Status on page 302</td>
</tr>
<tr>
<td>get_matched_subscription_datawriter_protocol_status_by_locator</td>
<td>7.3.6.5 OFFERED_DEADLINE_MISSED Status on page 303</td>
</tr>
<tr>
<td>get_liveliness_lost_status</td>
<td>7.3.6.6 OFFERED_INCOMPATIBLE_QOS Status on page 303</td>
</tr>
<tr>
<td>get_offered_deadline_missed_status</td>
<td>7.3.6.7 PUBLICATION_MATCHED Status on page 304</td>
</tr>
<tr>
<td>get_publication_match_status</td>
<td>7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305</td>
</tr>
<tr>
<td>get_reliable_writer_cache_changed_status</td>
<td>7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) on page 307</td>
</tr>
<tr>
<td>get_reliable_reader_activity_changed_status</td>
<td>7.3.6.10 SERVICE_REQUEST_ACCEPTED Status (DDS Extension) on page 308</td>
</tr>
<tr>
<td>get_service_request_accepted_status</td>
<td>A list of what changed in all of the above.</td>
</tr>
</tbody>
</table>

These methods are useful in the event that no Listener callback is set to receive notifications of status changes. If a Listener is used, the callback will contain the new status information, in which case calling these methods is unlikely to be necessary.
The `get_status_changes()` operation provides a list of statuses that have changed since the last time the status changes were ‘reset.’ A status change is reset each time the application calls the corresponding `get_*_status()`, as well as each time Connext DDS returns from calling the `Listener` callback associated with that status.

For more on status, see 7.3.4 Setting Up DataWriterListeners on page 294, 7.3.6 Statuses for DataWriters below, and 4.4 Listeners on page 189.

### 7.3.6 Statuses for DataWriters

There are several types of statuses available for a `DataWriter`. You can use the `get_*_status()` operations (7.3.15 Setting DataWriter QosPolicies on page 333) to access them, or use a `DataWriterListener` (7.3.4 Setting Up DataWriterListeners on page 294) to listen for changes in their values. Each status has an associated data structure and is described in more detail in the following sections.

- **7.3.6.1 APPLICATION_ACKNOWLEDGMENT_STATUS** below
- **7.3.6.2 DATA_WRITER_CACHE_STATUS** on the next page
- **7.3.6.3 DATA_WRITER_PROTOCOL_STATUS** on page 298
- **7.3.6.4 LIVELINESS_LOST Status** on page 302
- **7.3.6.5 OFFERED_DEADLINE_MISSED Status** on page 303
- **7.3.6.6 OFFERED_INCOMPATIBLE_QOS Status** on page 303
- **7.3.6.7 PUBLICATION.MATCHED Status** on page 304
- **7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status** (DDS Extension) on page 305
- **7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status** (DDS Extension) on page 307
- **7.3.6.10 SERVICE_REQUEST_ACCEPTED Status** (DDS Extension) on page 308

#### 7.3.6.1 APPLICATION_ACKNOWLEDGMENT_STATUS

This status indicates that a `DataWriter` has received an application-level acknowledgment for a DDS sample, and triggers a `DataWriter` callback:

```c
void DDSDataWriterListener::on_application_acknowledgment(
    DDSDataWriter * writer,
    const DDS_AcknowledgmentInfo & info)
```

`on_application_acknowledgment()` is called when a DDS sample is application-level acknowledged. It provides identities of the DDS sample and the acknowledging `DataReader`, as well as user-specified response data sent from the `DataReader` by the acknowledgment message—see Table 7.6 DDS_AcknowledgmentInfo.
### Table 7.6 DDS_AcknowledgmentInfo

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>subscription_handle</td>
<td>Subscription handle of the acknowledging DataReader.</td>
</tr>
<tr>
<td>struct DDS_SampleIdentity_t</td>
<td>sample_identity</td>
<td>Identity of the DDS sample being acknowledged.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>valid_response_data</td>
<td>Flag indicating validity of the user response data in the acknowledgment.</td>
</tr>
<tr>
<td>struct DDS_AckResponseData_t</td>
<td>response_data</td>
<td>User data payload of application-level acknowledgment message.</td>
</tr>
</tbody>
</table>

This status is only applicable when the DataWriter’s Reliability QosPolicy’s acknowledgment_kind is DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE or DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE.

#### 7.3.6.2 DATA_WRITER_CACHE_STATUS

This status keeps track of the number of DDS samples and instances in the DataWriter’s queue. For information on instance states, see 6.1 Instance States on page 258.

This status does not have an associated Listener. You can access this status by calling the DataWriter’s get_datawriter_cache_status() operation, which will return the status structure described in Table 7.7 DDS_DataWriterCacheStatus.

### Table 7.7 DDS_DataWriterCacheStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>sample_count_peak</td>
<td>Highest number of DDS samples in the DataWriter’s queue over the lifetime of the DataWriter.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>sample_count</td>
<td>Current number of DDS samples in the DataWriter’s queue. This number includes meta-samples, which represent the unregistration or disposal of an instance.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>alive_instance_count</td>
<td>Number of instances currently in the DataWriter’s queue that have an instance_state of ALIVE.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>alive_instance_count_peak</td>
<td>Highest number of ALIVE instances in the DataWriter’s queue over the lifetime of the DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>disposed_instance_count</td>
<td>Number of instances currently in the DataWriter’s queue that have an instance_state of NOT_ALIVE_DISPOSED.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>disposed_instance_count_peak</td>
<td>Highest number of NOT_ALIVE_DISPOSED instances in the DataWriter’s queue over the lifetime of the DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>unregistered_instance_count</td>
<td>Number of instances currently in the DataWriter’s queue that the DataWriter has unregistered from via the unregister_instance operation.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>unregistered_instance_count_peak</td>
<td>Highest number of instances that the DataWriter has unregistered from, over the lifetime of the DataWriter.</td>
</tr>
</tbody>
</table>
7.3.6 Statuses for Data Writers

7.3.6.3 DATA_WRITER_PROTOCOL_STATUS

This status includes internal protocol related metrics (such as the number of DDS samples pushed, pulled, filtered) and the status of wire-protocol traffic.

- **Pulled DDS samples** are DDS samples sent for repairs (that is, DDS samples that had to be resent), for late joiners, and all DDS samples sent by the local DataWriter when `push_on_write` (in 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389) is DDS_BOOLEAN_FALSE.
- **Pushed DDS samples** are DDS samples sent on `write()` when `push_on_write` is DDS_BOOLEAN_TRUE.
- **Filtered DDS samples** are DDS samples that are not sent due to DataWriter filtering (time-based filtering and ContentFilteredTopics), but this field is not currently supported.
- **DATA_FRAG messages** are fragments of samples if you are using DDS-level fragmentation. See 23.3 Large Data Fragmentation on page 970 for more information.

This status does not have an associated Listener. You can access this status by calling the following operations on the DataWriter (all of which return the status structure described in Table 7.8 DDS_DataWriter-ProtocolStatus):

- **get_datawriter_protocol_status()** returns the sum of the protocol status for all the matched subscriptions for the DataWriter.
- **get_matched_subscription_datawriter_protocol_status()** returns the protocol status of a particular matched subscription, identified by a subscription handle.
- **get_matched_subscription_datawriter_protocol_status_by_locator()** returns the protocol status of a particular matched subscription, identified by a locator. (See 15.2.1.1 Locator Format on page 770.)

**Note:** Status/data for a matched subscription is kept even if the DataReader is not active (has not responded to a heartbeat message with an ACK/NACK message after max_heartbeat_retries has been reached). The status/data will be removed only if the DataReader is gone: that is, the DataReader is destroyed and this change is propagated through a discovery update, or the DataReader's DomainParticipant is gone (either gracefully or its liveliness expired and Connext DDS is configured to purge not-alive participants). Once a matched DataReader is gone, its status is deleted. If you try to get the status/data for a matched DataReader that is gone, the 'get status' or 'get data' call will return an error.
### Table 7.8 DDS_DataWriterProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>pushed_sample_count</td>
<td>The number of user DDS samples pushed on write from this DataWriter to a matching DataReader. This field counts protocol (RTPS) messages pushed by a DataWriter when writing, unregistering, and disposing. The count is the number of sends done internally, and it may be greater than the number of user writes. This field counts whole samples, not fragments (in the case of large data). The fragment count is tracked in the pushed_fragment_count.</td>
</tr>
<tr>
<td></td>
<td>pushed_sample_count_change</td>
<td>Change in the pushed_sample_count since the last time the status was read.</td>
</tr>
<tr>
<td></td>
<td>pushed_sample_bytes</td>
<td>The number of bytes of user DDS samples pushed on write from this DataWriter to a matching DataReader. This field counts bytes of protocol (RTPS) messages pushed by a DataWriter when writing, unregistering, and disposing. The count of bytes corresponds to the number of sends done internally, and it may be greater than the number of user writes. When data fragmentation is used (for large data), this statistic is incremented as fragments are written.</td>
</tr>
<tr>
<td></td>
<td>pushed_sample_bytes_change</td>
<td>Change in pushed_sample_bytes since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>Not supported filtered_sample_count</td>
<td>The number of user samples preemptively filtered by this DataWriter due to ContentFilteredTopics.</td>
</tr>
<tr>
<td></td>
<td>Not supported filtered_sample_count_change</td>
<td>Change in the filtered_sample_count since the last time the status was read.</td>
</tr>
<tr>
<td></td>
<td>Not supported filtered_sample_bytes</td>
<td>The number of bytes of user samples preemptively filtered by this DataWriter due to ContentFilteredTopics.</td>
</tr>
<tr>
<td></td>
<td>Not supported filtered_sample_bytes_change</td>
<td>Change in the filtered_sample_bytes since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_heartbeat_count</td>
<td>The number of Heartbeats sent between this DataWriter and matching DataReaders.</td>
</tr>
<tr>
<td></td>
<td>sent_heartbeat_count_change</td>
<td>Change in the sent_heartbeat_count since the last time the status was read.</td>
</tr>
<tr>
<td></td>
<td>sent_heartbeat_bytes</td>
<td>The number of bytes of Heartbeats sent between this DataWriter and matching DataReaders.</td>
</tr>
<tr>
<td></td>
<td>sent_heartbeat_bytes_change</td>
<td>The incremental change in the number of bytes of Heartbeats sent between this DataWriter and matching DataReaders since the last time the status was read.</td>
</tr>
</tbody>
</table>
### Table 7.8 DDS_DataWriterProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>pulled_sample_count</td>
<td>The number of user DDS samples pulled from this DataWriter by matching DataReaders. When data fragmentation is used, this statistic is incremented as fragments are written.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>pulled_sample_count_change</td>
<td>Change in the <code>pulled_sample_count</code> since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>pulled_sample_bytes</td>
<td>The number of bytes of user DDS samples pulled from this DataWriter by matching DataReaders. When data fragmentation is used, this statistic is incremented as fragments are written.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>pulled_sample_bytes_change</td>
<td>Change in <code>pulled_sample_bytes</code> since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_ack_count</td>
<td>The number of ACKs from a DataReader received by this DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_ack_count_change</td>
<td>Change in the <code>received_ack_count</code> since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_ack_bytes</td>
<td>The number of bytes of ACKs from a DataReader received by this DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_ack_bytes_change</td>
<td>Change in <code>received_ack_bytes</code> since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_nack_count</td>
<td>The number of NACKs from a DataReader received by this DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_nack_count_change</td>
<td>Change in the <code>received_nack_count</code> since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_nack_bytes</td>
<td>The number of bytes of NACKs from a DataReader received by this DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_nack_bytes_change</td>
<td>Change in <code>received_nack_bytes</code> since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_gap_count</td>
<td>The number of GAPs sent from this DataWriter to matching DataReaders.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_gap_count_change</td>
<td>Change in the <code>sent_gap_count</code> since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_gap_bytes</td>
<td>The number of bytes of GAPs sent from this DataWriter to matching DataReaders.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_gap_bytes_change</td>
<td>Change in <code>sent_gap_bytes</code> since the last time the status was read.</td>
</tr>
</tbody>
</table>
### Table 7.8 DDS_DataWriterProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td><strong>Not supported</strong> rejected_sample_count</td>
<td>These fields are not supported.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>send_window_size</td>
<td>Current size of the send window (maximum number of outstanding DDS samples allowed in the DataWriter's queue), as determined by the min/max_send_window_size fields in Table 7.45 DDS_RtpsReliableWriterProtocol_t. (See 7.5.5.4 Configuring the Send Window Size on page 397 for information on how the send window size might change.)</td>
</tr>
<tr>
<td></td>
<td>pushed_fragment_count</td>
<td>The number of fragments (DATA_FRAG messages) that have been pushed from this DataWriter to a DataReader. This count is incremented as each DATA_FRAG message is sent, not when the entire sample has been sent. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td></td>
<td>pushed_fragment_bytes</td>
<td>The number of bytes of DATA_FRAG messages that have been pushed by this DataWriter. This statistic is incremented as each DATA_FRAG message is sent, not when the entire sample has been sent. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>pulled_fragment_count</td>
<td>The number of fragments (DATA_FRAG messages) that have been pulled from this DataWriter by a DataReader. This count is incremented as each DATA_FRAG message is sent, not when the entire sample has been sent. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td></td>
<td>pulled_fragment_bytes</td>
<td>The number of bytes of DATA_FRAG messages that have been pulled from this DataWriter by a DataReader. This statistic is incremented as each DATA_FRAG message is sent, not when the entire sample has been sent. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_nack_fragment_count</td>
<td>The number of NACK_FRAG messages that have been received by this DataWriter. NACK FRAG RTPS messages are sent when large data is used in conjunction with reliable communication. They have the same properties as NACK messages, but instead of applying to samples, they apply to fragments. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td></td>
<td>received_nack_fragment_bytes</td>
<td>The number of bytes of NACK_FRAG messages that have been received by this DataWriter. NACK FRAG RTPS messages are sent when large data is used in conjunction with reliable communication. They have the same properties as NACK messages, but instead of applying to samples, they apply to fragments. Applicable only when data is fragmented.</td>
</tr>
</tbody>
</table>
### Table 7.8 DDS_DataWriterProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_SequenceNumber_</td>
<td>first_available_sample_sequence_number</td>
<td>Sequence number of the first available DDS sample in the DataWriter’s reliability queue.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_</td>
<td>last_available_sample_sequence_number</td>
<td>Sequence number of the last available DDS sample in the DataWriter’s reliability queue.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_</td>
<td>first_unacknowledged_sample_sequence_number</td>
<td>Sequence number of the first unacknowledged DDS sample in the DataWriter’s reliability queue.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_</td>
<td>first_available_sample_virtual_sequence_number</td>
<td>Virtual sequence number of the first available DDS sample in the DataWriter’s reliability queue.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_</td>
<td>last_available_sample_virtual_sequence_number</td>
<td>Virtual sequence number of the last available DDS sample in the DataWriter’s reliability queue.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_</td>
<td>first_unacknowledged_sample_virtual_sequence_number</td>
<td>Virtual sequence number of the first unacknowledged DDS sample in the DataWriter’s reliability queue.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_</td>
<td>first_unelapsed_keep_duration_sample_sequence_number</td>
<td>Instance Handle of the matching remote DataReader for which the DataWriter has kept the first available DDS sample in the reliability queue.</td>
</tr>
</tbody>
</table>

#### 7.3.6.4 LIVELINESS_LOST Status

A change to this status indicates that the DataWriter failed to signal its liveliness within the time specified by the 7.5.15 LIVELINESS QosPolicy on page 427.

It is different than the 7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) on page 307 status that provides information about the liveliness of a DataWriter’s matched DataReaders; this status reflects the DataWriter’s own liveliness.

The structure for this status appears in Table 7.9 DDS_LivelinessLostStatus.
Table 7.9 DDS_LivelinessLostStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative number of times the DataWriter failed to explicitly signal its liveliness within the liveliness period.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>total_count_change</td>
<td>The change in total_count since the last time the Listener was called or the status was read.</td>
</tr>
</tbody>
</table>

The DataWriterListener’s on_liveliness_lost() callback is invoked when this status changes. You can also retrieve the value by calling the DataWriter’s get_liveliness_lost_status() operation.

7.3.6.5 OFFERED_DEADLINE_MISSED Status

A change to this status indicates that the DataWriter failed to write data within the time period set in its 7.5.7 DEADLINE QosPolicy on page 405.

The structure for this status appears in Table 7.10 DDS_OfferedDeadlineMissedStatus.

Table 7.10 DDS_OfferedDeadlineMissedStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative number of times the DataWriter failed to write within its offered deadline.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>total_count_change</td>
<td>The change in total_count since the last time the Listener was called or the status was read.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>last_instance_handle</td>
<td>Handle to the last data-instance in the DataWriter for which an offered deadline was missed.</td>
</tr>
</tbody>
</table>

The DataWriterListener’s on_offered_deadline_missed() operation is invoked when this status changes. You can also retrieve the value by calling the DataWriter’s get_deadline_missed_status() operation.

7.3.6.6 OFFERED_INCOMPATIBLE_QOS Status

A change to this status indicates that the DataWriter discovered a DataReader for the same Topic, but that DataReader had requested QoS settings incompatible with this DataWriter’s offered QoS.

The structure for this status appears in Table 7.11 DDS_OfferedIncompatibleQoSStatus.

Table 7.11 DDS_OfferedIncompatibleQoSStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative number of times the DataWriter discovered a DataReader for the same Topic with a requested QoS that is incompatible with that offered by the DataWriter.</td>
</tr>
</tbody>
</table>
### 7.3.6 Statuses for DataWriters

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count_change</td>
<td>The change in total_count since the last time the <code>Listener</code> was called or the status was read.</td>
</tr>
<tr>
<td>DDS_QosPolicyId_t</td>
<td>last_policy_id</td>
<td>The ID of the QosPolicy that was found to be incompatible the last time an incompatibility was detected. (Note: if there are multiple incompatible policies, only one of them is reported here.)</td>
</tr>
<tr>
<td>DDS_QosPolicyCountSeq</td>
<td>policies</td>
<td>A list containing—for each policy—the total number of times that the <code>DataWriter</code> discovered a <code>DataReader</code> for the same <code>Topic</code> with a requested QoS that is incompatible with that offered by the <code>DataWriter</code>.</td>
</tr>
</tbody>
</table>

The `DataWriterListener`’s `on_offered_incompatible_qos()` callback is invoked when this status changes. You can also retrieve the value by calling the `DataWriter`’s `get_offered_incompatible_qos_status()` operation.

#### 7.3.6.7 PUBLICATION_MATCHED Status

A change to this status indicates that the `DataWriter` discovered a matching `DataReader`.

A ‘match’ occurs only if the `DataReader` and `DataWriter` have the same `Topic`, same or compatible data type, and compatible QosPolicies. (For more information on compatible data types, see the RTI Connext DDS Core Libraries Extensible Types Guide.) In addition, if user code has directed Connext DDS to ignore certain `DataReaders`, then those `DataReaders` will never be matched. See 18.4.2 Ignoring Publications and Subscriptions on page 843 for more on setting up a `DomainParticipant` to ignore specific `DataReaders`.

This status is also changed (and the listener, if any, called) when a match is ended. A local `DataWriter` will become "unmatched" from a remote `DataReader` when that `DataReader` goes away for any of the following reasons:

- The matched `DataReader's DomainParticipant` has lost liveliness.
- This `DataWriter` or the matched `DataReader` has changed QoS such that the entities are now incompatible.
- The matched `DataReader` has been deleted.

This status may reflect changes from multiple match or unmatch events, and the current_count_change can be used to determine the number of changes since the listener was called back or the status was checked.

The structure for this status appears in Table 7.12 DDS_PublicationMatchedStatus.
### Table 7.12 DDS_PublicationMatchedStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative number of times that this DataWriter discovered a &quot;match&quot; with a DataReader. This number increases whenever a new match is discovered. It does not decrease when an existing match goes away for any of the reasons listed above.</td>
</tr>
<tr>
<td></td>
<td>total_count_change</td>
<td>The changes in total_count since the last time the listener was called or the status was read. Note that this number will never be negative (because it’s the total number of times the DataWriter ever matched with a DataReader).</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>current_count</td>
<td>The number of DataReaders currently matched to the DataWriter. This number increases when a new match is discovered and decreases when an existing match goes away for any of the reasons listed above.</td>
</tr>
<tr>
<td></td>
<td>current_count_peak</td>
<td>Greatest number of DataReaders that matched this DataWriter simultaneously. That is, there was no moment in time when more than this many DataReaders matched this DataWriter. (As a result, total_count can be higher than current_count_peak.)</td>
</tr>
<tr>
<td></td>
<td>current_count_change</td>
<td>The change in current_count since the last time the listener was called or the status was read. Note that a negative current_count_change means that one or more DataReaders have become unmatched for one or more of the reasons listed above.</td>
</tr>
</tbody>
</table>

The DataWriterListener’s on_publication_matched() callback is invoked when this status changes. You can also retrieve the value by calling the DataWriter’s get_publication_match_status() operation.

### 7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension)

A change to this status indicates that the number of unacknowledged DDS samples\(^1\) in a reliable DataWriter's cache has reached one of these trigger points:

- The cache is empty (contains no unacknowledged DDS samples)
- The cache is full (the number of unacknowledged DDS samples has reached the value specified in DDS_ResourceLimitsQosPolicy::max_samples)

\(^1\)If batching is enabled, this still refers to a number of DDS samples, not batches.
The number of unacknowledged DDS samples has reached a high or low watermark. See the `high_watermark` and `low_watermark` fields in Table 7.45 DDS_RtpsReliableWriterProtocol_t of the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

For more about the reliable protocol used by Connext DDS and specifically, what it means for a DDS sample to be ‘unacknowledged,’ see Reliable Communications (Chapter 11 on page 688).

The structure for this status appears in Table 7.13 DDS_ReliableWriterCacheChangedStatus. The supporting structure, DDS_ReliableWriterCacheEventCount, is described in Table 7.14 DDS_ReliableWriterCacheEventCount.

### Table 7.13 DDS_ReliableWriterCacheChangedStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_ReliableWriterCacheEventCount</td>
<td>empty_reliable_writer_cache</td>
<td>How many times the reliable DataWriter's cache of unacknowledged DDS samples has become empty.</td>
</tr>
<tr>
<td>DDS_ReliableWriterCacheEventCount</td>
<td>full_reliable_writer_cache</td>
<td>How many times the reliable DataWriter's cache of unacknowledged DDS samples has become full.</td>
</tr>
<tr>
<td>DDS_ReliableWriterCacheEventCount</td>
<td>low_watermark_reliable_writer_cache</td>
<td>How many times the reliable DataWriter's cache of unacknowledged DDS samples has fallen to the low watermark.</td>
</tr>
<tr>
<td>DDS_ReliableWriterCacheEventCount</td>
<td>high_watermark_reliable_writer_cache</td>
<td>How many times the reliable DataWriter's cache of unacknowledged DDS samples has risen to the high watermark.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>unacknowledged_sample_count</td>
<td>The current number of unacknowledged DDS samples in the DataWriter's cache.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>unacknowledged_sample_count_peak</td>
<td>The highest value that unacknowledged_sample_count has reached until now.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>replaced_unacknowledged_sample_count</td>
<td>Total number of unacknowledged samples that have been replaced by a DataWriter after applying the KEEP_LAST setting in the 7.5.12 HISTORY QosPolicy on page 420 policy.</td>
</tr>
</tbody>
</table>

### Table 7.14 DDS_ReliableWriterCacheEventCount

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>The total number of times the event has occurred.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>total_count_change</td>
<td>The number of times the event has occurred since the Listener was last invoked or the status read.</td>
</tr>
</tbody>
</table>
7.3.6 Statuses for DataWriters

The DataWriterListener’s on_reliable_writer_cache_changed() callback is invoked when this status changes. You can also retrieve the value by calling the DataWriter’s get_reliable_writer_cache_changed_status() operation.

If a reliable DataWriter’s send window is finite, with both RtpsReliableWriterProtocol_t.min_send_window_size and RtpsReliableWriterProtocol_t.max_send_window_size set to positive values, then full_reliable_writer_cache_status counts the number of times the unacknowledged DDS sample count reaches the send window size.

7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension)

This status indicates that one or more reliable DataReaders has become active or inactive.

This status is the reciprocal status to the 8.3.7.4 LIVELINESS_CHANGED Status on page 524 on the DataReader. It is different than 7.3.6.4 LIVELINESS_LOST Status on page 302 status on the DataWriter, in that the latter informs the DataWriter about its own liveness; this status informs the DataWriter about the liveness of its matched DataReaders.

A reliable DataReader is considered active by a reliable DataWriter with which it is matched if that DataReader acknowledges the DDS samples that it has been sent in a timely fashion. For the definition of "timely" in this context, see 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

This status is only used for DataWriters whose 7.5.21 RELIABILITY QosPolicy on page 447 is set to RELIABLE. For best-effort DataWriters, all counts in this status will remain at zero.

The structure for this status appears in Table 7.15 DDS_ReliableReaderActivityChangedStatus.

Table 7.15 DDS_ReliableReaderActivityChangedStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>active_count</td>
<td>The current number of reliable readers currently matched with this reliable DataWriter.</td>
</tr>
<tr>
<td></td>
<td>inactive_count</td>
<td>The number of reliable readers that have been dropped by this reliable DataWriter because they failed to send acknowledgments in a timely fashion.</td>
</tr>
<tr>
<td></td>
<td>active_count_change</td>
<td>The change in the number of active reliable DataReaders since the Listener was last invoked or the status read.</td>
</tr>
<tr>
<td></td>
<td>inactive_count_change</td>
<td>The change in the number of inactive reliable DataReaders since the Listener was last invoked or the status read.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>last_instance_handle</td>
<td>The instance handle of the last reliable DataReader to be determined to be inactive.</td>
</tr>
</tbody>
</table>

The DataWriterListener’s on_reliable_reader_activity_changed() callback is invoked when this status changes. You can also retrieve the value by calling the DataWriter’s get_reliable_reader_activity_changed_status() operation.
7.3.6.10 SERVICE_REQUEST_ACCEPTED Status (DDS Extension)

A change to this status indicates that ServiceRequest for the TopicQuery service is dispatched to this DataWriter for processing. For more information, see Topic Queries (Chapter 24 on page 977).

The structure for this status appears in Table 7.16 DDS_ServiceRequestAcceptedStatus.

The DataWriterListener’s on_service_request_accepted() callback is invoked when this status changes.

You can also retrieve the value by calling the DataWriter’s get_service_request_accepted_status() operation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>The total cumulative number of ServiceRequests that have been accepted by a DataWriter.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>total_count_change</td>
<td>The incremental changes in total_count since the last time the listener was called or the status was read.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>current_count</td>
<td>The current number of ServiceRequests that have been accepted by this DataWriter.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>last_request_handle</td>
<td>A handle to the last DDS_ServiceRequest that caused the DataWriter's status to change.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>service_id</td>
<td>ID of the service to which the accepted Request belongs</td>
</tr>
</tbody>
</table>

7.3.7 Using a Type-Specific DataWriter (FooDataWriter)

(Note: This section does not apply to the Modem C++ API, where a DataWriter's data type is part of its template definition: DataWriter<Foo>)

Recall that a Topic is bound to a data type that specifies the format of the data associated with the Topic. Data types are either defined dynamically or in code generated from definitions in IDL or XML; see Data Types and DDS Data Samples (Chapter 3 on page 27). For each of your application's generated data types, such as 'Foo', there will be a FooDataWriter class (or a set of functions in C). This class allows the application to use a type-safe interface to interact with DDS samples of type 'Foo'. You will use the FooDataWriter's write() operation used to send data. For dynamically defined data-types, you will use the DynamicDataWriter class.

In fact, you will use the FooDataWriter any time you need to perform type-specific operations, such as registering or writing instances. Table 7.3 DataWriter Operations indicates which operations must be called using FooDataWriter. For operations that are not type-specific, you can call the operation using either a FooDataWriter or a DDSDataWriter object\(^1\).

\(^1\)In the C API, the non type-specific operations must be called using a DDS_DataWriter pointer.
You may notice that the Publisher’s `create_datawriter()` operation returns a pointer to an object of type `DDSDataWriter`; this is because the `create_datawriter()` method is used to create `DataWriters` of any data type. However, when executed, the function actually returns a specialization (an object of a derived class) of the `DataWriter` that is specific for the data type of the associated `Topic`. For a `Topic` of type ‘Foo’, the object actually returned by `create_datawriter()` is a `FooDataWriter`.

To safely cast a generic `DDSDataWriter` pointer to a `FooDataWriter` pointer, you should use the static `narrow()` method of the `FooDataWriter` class. The `narrow()` method will return NULL if the generic `DDSDataWriter` pointer is not pointing at an object that is really a `FooDataWriter`.

For instance, if you create a `Topic` bound to the type ‘Alarm’, all `DataWriters` created for that `Topic` will be of type ‘AlarmDataWriter.’ To access the type-specific methods of `AlarmDataWriter`, you must cast the generic `DDSDataWriter` pointer returned by `create_datawriter()`.

```cpp
DDSDataWriter* writer = publisher->create_datawriter(  
    topic, writer_qos, NULL, NULL);
AlarmDataWriter *alarm_writer = AlarmDataWriter::narrow(writer);
if (alarm_writer == NULL) {
    // ... error
};
```

In the C API, there is also a way to do the opposite of `narrow`. `FooDataWriter_as_datawriter()` casts a `FooDataWriter` as a `DDSDataWriter`, and `FooDataReader_as_datareader()` casts a `FooDataReader` as a `DDSDataReader`.

### 7.3.8 Writing Data

The `write()` operation informs `Connext DDS` that there is a new value for a data-instance to be published for the corresponding `Topic`. By default, calling `write()` will send the data immediately over the network (assuming that there are matched `DataReaders`). However, you can configure and execute operations on the `DataWriter`’s `Publisher` to buffer the data so that it is sent in a batch with data from other `DataWriters` or even to prevent the data from being sent. Those sending “modes” are configured using the 7.4.6 `PRESENTATION` `QosPolicy` on page 362 as well as the Publisher’s `suspend/resume_publications()` operations. The actual transport-level communications may be done by a separate, lower-priority thread when the `Publisher` is configured to send the data for its `DataWriters`. For more information on threads, see `Connext DDS Threading Model` (Chapter 21 on page 908).

When you call `write()`, `Connext DDS` automatically attaches a stamp of the current time that is sent with the DDS data sample to the `DataReader(s)`. The timestamp appears in the `source_timestamp` field of the `DDS_SampleInfo` structure that is provided along with your data using `DataReaders` (see 8.4.6 `The SampleInfo Structure` on page 565).

```cpp
DDS_ReturnCode_t write (const Foo &instance_data,  
    const DDS_InstanceHandle_t &handle)
```

You can use an alternate `DataWriter` operation called `write_w_timestamp()`. This performs the same action as `write()`, but allows the application to explicitly set the `source_timestamp`. This is useful when
you want the user application to set the value of the timestamp instead of the default clock used by Connext DDS.

```
DDS_ReturnCode_t write_w_timestamp (  
    const Foo &instance_data, 
    const DDS_InstanceHandle_t &handle, 
    const DDS_Time_t &source_timestamp)
```

Note that, in general, the application should not mix these two ways of specifying timestamps. That is, for each `DataWriter`, the application should either always use the automatic timestamping mechanism (by calling the normal operations) or always specify a timestamp (by calling the “w_timestamp” variants of the operations). Mixing the two methods may result in not receiving sent data.

You can also use an alternate `DataWriter` operation, `write_w_params()`, which performs the same action as `write()`, but allows the application to explicitly set the fields contained in the `DDS_WriteParams` structure, see Table 7.17 `DDS_WriteParams_t`.

### Table 7.17 DDS_WriteParams_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>replace_auto</td>
<td>Allows retrieving the actual value of those fields that were automatic. When this field is set to true, the fields that were configured with an automatic value (for example, DDS_AUTO_SAMPLE_IDENTITY in identity) receive their actual value after <code>write_w_params</code> is called.</td>
</tr>
</tbody>
</table>
| DDS_SampleIdentity_t | identity | Identity of the DDS sample being written. The identity consists of a pair (Virtual Writer GUID, Virtual Sequence Number). When the value DDS_AUTO_SAMPLE_IDENTITY is used, the `write_w_params()` operation will determine the DDS sample identity as follows:  
  * The Virtual Writer GUID (writer_guid) is the virtual GUID associated with the `DataWriter` writing the DDS sample. This virtual GUID is configured using the member `virtual_guid` in 7.3.6.3 `DATA_WRITER_PROTOCOL_STATUS` on page 298.  
  * The Virtual Sequence Number (sequence_number) is increased by one with respect to the previous value. The virtual sequence numbers for a given virtual GUID must be strictly monotonically increasing. If you try to write a DDS sample with a sequence number smaller or equal to the last sequence number, the write operation will fail. |

*A DataReader can inspect the identity of a received DDS sample by accessing the fields `original_publication_virtual_guid` and `original_publication_virtual_sequence_number` in 8.4.6 `The SampleInfo Structure` on page 565.*
### Table 7.17 DDS_WriteParams_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_SampleIdentity_t</td>
<td>related_sample_identity</td>
<td>The identity of another DDS sample related to this one. The value of this field identifies another DDS sample that is logically related to the one that is written. For example, the DataWriter created by a Replier (sets Introduction to the Request-Reply Communication Pattern (Chapter 26 on page 1000)) uses this field to associate the identity of the DDS request sample to reponse sample. To specify that there is no related DDS sample identity use the value DDS_UNKNOWN_SAMPLE.IDENTITY, A DataReader can inspect the related DDS sample identity of a received DDS sample by accessing the fields related_original_publication_virtual_guid and related_original_publication_virtual_sequence_number in 8.4.6 The SampleInfo Structure on page 565.</td>
</tr>
<tr>
<td>DDS_Time</td>
<td>source_timestamp</td>
<td>Source timestamp that will be associated to the DDS sample that is written. If source_timestamp is set to DDS_TIMER_INVALID, the middleware will assign the value. A DataReader can inspect the source_timestamp value of a received DDS sample by accessing the field source_timestamp 8.4.6 The SampleInfo Structure on page 565.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>handle</td>
<td>The instance handle. This value can be either the handle returned by a previous call to register_instance() or the special value DDS_HANDLE NIL.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>priority</td>
<td>Positive integer designating the relative priority of the DDS sample, used to determine the transmission order of pending transmissions. To use publication priorities, the DataWriter's 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444 must be set for asynchronous publishing and the DataWriter must use a FlowController with a highest-priority first scheduling policy. For Multi-channel DataWriters, the publication priority of a DDS sample may be used as a filter criteria for determining channel membership. For more information, see 7.6.4 Prioritized DDS Samples on page 476.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>flag</td>
<td>Flags for the DDS sample, represented as a 32-bit integer, of which only the 16 least-significant bits are used. RTI reserves least-significant bits [0-7] for middleware-specific usage. The application can use least significant bits [8-15]. An application can inspect the flags associated with a received DDS sample by checking the flag field in 8.4.6 The SampleInfo Structure on page 565. For details about the reserved bits see 8.4.6 The SampleInfo Structure on page 565. Default 0 (no flags are set).</td>
</tr>
<tr>
<td>struct DDS_GUID_t</td>
<td>source_guid</td>
<td>Identifies the application logical data source associated with the sample being written.</td>
</tr>
<tr>
<td>struct DDS_GUID_t</td>
<td>related_source_guid</td>
<td>Identifies the application logical data source that is related to the sample being written.</td>
</tr>
<tr>
<td>struct DDS_GUID_t</td>
<td>related_reader_guid</td>
<td>Identifies a DataReader that is logically related to the sample that is being written.</td>
</tr>
</tbody>
</table>
When using the C API, a newly created variable of type DDS_WriteParams_t should be initialized by setting it to DDS_WRITEPARAMS_DEFAULT.

The write() operation also asserts liveliness on the DataWriter, the associated Publisher, and the associated DomainParticipant. It has the same effect with regards to liveliness as an explicit call to assert_liveliness(), see 7.3.17 Asserting Liveliness on page 344 and the 7.5.15 LIVELINESS QosPolicy on page 427. Maintaining liveliness is important for DataReaders to know that the DataWriter still exists and for the proper behavior of the 7.5.17 OWNERSHIP QosPolicy on page 434.

See also: 9.6 Clock Selection on page 677.

7.3.8.1 Blocking During a write()

The write() operation may block if the 7.5.21 RELIABILITY QosPolicy on page 447 kind is set to Reliable, the send window is full, or the modification would cause data to be lost. Specifically, write() may block in the following situations (note that the list may not be exhaustive):

- If the send window is specified (max/min_send_window_size fields in the DDS_RtpsReliableWriterProtocol_t structure in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 are not LENGTH_UNLIMITED) and the send window is full. Blocking in this case occurs with both KEEP_LAST and KEEP_ALL history kinds.

- If max_samples or max_samples_per_instance in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 (or max_batches in 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401) are exceeded and none of the samples can be replaced because they are not fully ACKed. Blocking in this case only applies to the KEEP_ALL history kind.

This operation may also block when using BEST_EFFORT Reliability 7.5.21 RELIABILITY QosPolicy on page 447 and ASYNCHRONOUS Publish Mode (7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444) QoS settings. In this case, the DataWriter will queue DDS samples until they are sent by the asynchronous publishing thread. The number of DDS samples that can be stored is determined by the 7.5.12 HISTORY QosPolicy on page 420. If the asynchronous thread does not send DDS samples fast enough (such as when using a slow FlowController (7.6 FlowControllers (DDS Extension) on page 470)), the queue may fill up. In that case, subsequent write calls will block.

If this operation does block for any of the above reasons, the 7.5.21 RELIABILITY QosPolicy on page 447's max_blocking_time configures the maximum time the write operation may block (waiting for space to become available). If max_blocking_time elapses before the DataWriter can store the modification without exceeding the limits, the operation will fail and return RETCODE_TIMEOUT for KEEP_ALL configurations.

7.3.8.2 write() behavior with KEEP_LAST and KEEP_ALL

Following is how the write operation behaves when KEEP_LAST (in the 7.5.12 HISTORY QosPolicy on page 420) and RELIABLE (in the 7.5.21 RELIABILITY QosPolicy on page 447) are used:
The send window size is determined by the `max/min_send_window_size` fields in the DDS_RtpsReliableWriterProtocol_t structure in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389. If a send window is specified (`max_send_window_size` is not UNLIMITED) and the window is full, the operation will block until one of the samples in the send window is protocol-acknowledged (ACKed) (Note 1) or until the `max_blocking_time` in the 7.5.21 RELIABILITY QosPolicy on page 447 (`writer_qos.reliability.max_blocking_time`) expires.

Then, the `DataWriter` will try to add the new sample to the writer history.

If the instance associated with the sample is present in the writer history and there are `depth` (in the 7.5.12 HISTORY QosPolicy on page 420) samples in the instance, the `DataWriter` will replace the oldest sample of that instance independently of that sample’s acknowledged status, and the write operation will return DDS_RETCODE_OK. Otherwise, no sample will be replaced and the write operation will continue.

If the instance associated with the sample is not present in the writer history and `max_instances` (in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451) is exceeded, the `DataWriter` will try to replace an existing instance (and its samples) according to the value of the instance_replacement field in the 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 401 (see 7.5.6.1 Configuring DataWriter Instance Replacement on page 404).

- If no instance can be replaced, the write operation returns a DDS_RETCODE_OUT_OF_RESOURCES error.

If `max_samples` (in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451) is exceeded, the `DataWriter` will try to drop a sample from a different instance as follows:

- The `DataWriter` will try first to remove a fully ACKed (Note 2) sample from a different instance 'I' as long as that sample is not the last remaining sample for the instance 'I'. To find this sample, the `DataWriter` starts iterating from the oldest sample in the writer history to the newest sample.
- If no such sample is found, the `DataWriter` will replace the oldest sample in the writer history.
- The sample is added to the writer history, and the write operation returns DDS_RETCODE_OK.

Following is how the write operation behaves when KEEP_ALL (in the 7.5.12 HISTORY QosPolicy on page 420) and RELIABLE (in the 7.5.21 RELIABILITY QosPolicy on page 447) are used:

- The send window size is determined by the `max/min_send_window_size` fields in the DDS_RtpsReliableWriterProtocol_t structure in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389. If a send window is specified (`max_send_window_size` is not UNLIMITED) and the window is full, the operation will block until one of the samples in the send window is protocol-ACKed (Note 1) or until the `max_blocking_time` in the 7.5.21 RELIABILITY QosPolicy on page 447 (`writer_qos.reliability.max_blocking_time`) expires.
7.3.8 Writing Data

- If `writer_qos.reliability.max_blocking_time` expires, the write operation returns DDS_RETCODE_TIMEOUT.

- When a sample is protocol-ACKed (Note 1) before `max_blocking_time` expires, the DataWriter will try to add the sample to the writer history as follows:
  - If the instance associated with the sample is not present in the writer history and `max_instances` is exceeded, the DataWriter will try to replace an existing instance (and its samples) according to the value of the `instance_replacement` field in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401 (see 7.5.6.1 Configuring DataWriter Instance Replacement on page 404).
    - If no instance can be replaced, the write operation returns a DDS_RETCODE_OUT_OF_RESOURCES error.
  
- If `max_samples` is exceeded, the DataWriter will go through the samples in the order in which they were added, and it will replace the first sample that is fully ACKed (Note 2).
  - If no fully ACKed sample is found, the DataWriter will block (Note 3) until a sample is fully ACKed and can be replaced or `writer_qos.reliability.max_blocking_time` expires. If `writer_qos.reliability.max_blocking_time` expires, the write operation will return DDS_RETCODE_TIMEOUT.

- If `max_samples_per_instance` is exceeded, the DataWriter will go through the samples of the instance in the order in which they were added, and it will replace the first sample that is fully ACKed.
  - If no fully ACKed sample is found, the DataWriter will block (Note 3) until a sample is fully ACKed and can be replaced or `writer_qos.reliability.max_blocking_time` expires. If `writer_qos.reliability.max_blocking_time` expires, the write operation will return DDS_RETCODE_TIMEOUT.

- The sample is added to the writer history, and the write operation returns DDS_RETCODE_OK.

See 7.3.12.1 Application Acknowledgment Kinds on page 318 for more information on the following notes:

**Note 1:** A sample in the writer history is considered “protocol ACKed” when the sample has been individually ACKed at the RTPS protocol level by each one of the DataReaders that matched the DataWriter at the moment the sample was added to the DataWriter queue.

- Late joiners do not change the protocol ACK state of a sample. If a sample is marked as protocol ACKed because it has been acknowledged by all the matching DataReaders and a DataReader joins later on, the historical sample is still considered protocol ACKed even if it has not been received by the late joiner.
If a sample 'S1' is protocol ACKed and a TopicQuery is received, triggering the publication of 'S1', the sample is still considered protocol ACKed. If a sample 'S1' is not ACKed and a TopicQuery is received triggering the publication of 'S1', the DataWriter will require that both the matching DataReaders on the live RTPS channel and the DataReader on the TopicQuery channel individually protocol ACK the sample in order to consider the sample protocol ACKed.

**Note 2:** A sample in the writer history is considered “fully ACKed” when all of the following conditions are met:

- The sample is protocol-ACKed.
- The sample has been “application-level ACKed” by all the DataReaders matching the DataWriter that have their reader_qos.reliability.acknowledgment_kind set to AUTO_ACKNOWLEDGMENT_MODE or EXPLICIT_ACKNOWLEDGMENT_MODE. Once the sample is application-level ACKed, it cannot change its status to not ACKed after new DataReaders are matched. (Application-level ACK occurs when the application acknowledges receipt of a sample.)
- If required subscriptions are enabled (see 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370), the sample must also be ACKed by all the required subscriptions configured on the DataWriter.

**Note 3:** It is possible within a single call to the write operation for a DataWriter to block both when the send window is full and then again when max_samples or max_samples_per_instance is exceeded. This can happen because blocking on the send window only considers protocol-ACKed samples, while blocking based on resource limits considers fully-ACKed samples. In any case, the total max blocking time of a single call to the write operation will not exceed writer_qos.reliability.max_blocking_time.

The write operation on a DataWriter configured to use batching may also block if the sample being written cannot be added to the existing outstanding batch and the batch has to be synchronously flushed within the context of the write thread (see 7.5.2.1 Synchronous and Asynchronous Flushing on page 376). The flushing operation may block under the same scenarios described above for individual samples, taking into account that the send window is applied per batch and not per sample.

The unregister_instance() and dispose() operations, with regards to KEEP_LAST and KEEP_ALL, behave the same as for the write() operation. See 7.3.14.2 Registering Instances on page 326, 7.3.14.4 Unregistering Instances on page 328, and 7.3.14.3 Disposing Instances on page 328.

### 7.3.9 Flushing Batches of DDS Data Samples

The flush() operation makes a batch of DDS data samples available to be sent on the network.
If the `DataWriter`’s `PUBLISH_MODE QosPolicy (DDS Extension)` on page 444 `kind` is not **ASYNCHRONOUS**, the batch will be sent on the network immediately in the context of the calling thread.

If the `DataWriter`’s publish mode `QosPolicy` `kind` is **ASYNCHRONOUS**, the batch will be sent in the context of the asynchronous publishing thread.

The `flush()` operation may block based on the conditions described in 7.3.8.1 **Blocking During a write()** on page 312.

If this operation does block, the `max_blocking_time` in the `RELIABILITY QosPolicy` on page 447 configures the maximum time the write operation may block (waiting for space to become available). If `max_blocking_time` elapses before the `DataWriter` is able to store the modification without exceeding the limits, the operation will fail and return `TIMEOUT`.

For more information on batching, see the 7.5.2 **BATCH QosPolicy (DDS Extension)** on page 374.

### 7.3.10 Writing Coherent Sets of DDS Data Samples

A publishing application can request that a set of DDS data-sample changes be propagated in such a way that they are interpreted at the receivers’ side as a cohesive set of modifications. In this case, the receiver will only be able to access the data after all the modifications in the set are available at the subscribing end.

This is useful in cases where the values are inter-related. For example, suppose you have two data-instances representing the ‘altitude’ and ‘velocity vector’ of the same aircraft. If both are changed, it may be important to ensure that reader see both together (otherwise, it may erroneously interpret that the aircraft is on a collision course).

To use this mechanism in C, Traditional C++, Java and .NET:

1. Call the `Publisher`’s `begin_coherent_changes()` operation to indicate the start a coherent set.
2. For each DDS sample in the coherent set: call the `FooDataWriter`’s `write()` operation.
3. Call the `Publisher`’s `end_coherent_changes()` operation to terminate the set.

In the Modern C++ API:

1. Instantiate a `dds::pub::CoherentSet` passing a publisher to the constructor
2. For each DDS sample in the coherent set call `dds::pub::DataWriter<Foo>::write()`.
3. Let the `dds::pub::CoherentSet` destructor terminate the set or explicitly call `dds::pub::CoherentSet::end()`
Calls to `begin_coherent_changes()` and `end_coherent_changes()` can be nested. Publisher’s samples (samples published by any of the DataWriters within the Publisher) that are not published within a `begin_coherent_changes/end_coherent_changes` block will not be provided to the DataReaders as a set.

See also: the coherent_access field in the 7.4.6 PRESENTATION QosPolicy on page 362 and the coherent_set_info field in 8.4.6 The SampleInfo Structure on page 565.

### 7.3.11 Waiting for Acknowledgments in a DataWriter

The DataWriter’s `wait_for_acknowledgments()` operation blocks the calling thread until either all data written by the reliable DataWriter is acknowledged by (a) all reliable DataReaders that are matched and alive and (b) by all required subscriptions (see 7.3.13 Required Subscriptions on page 323), or until the duration specified by the max_wait parameter elapses, whichever happens first.

Note that if a thread is blocked in the call to `wait_for_acknowledgments()` on a DataWriter and a different thread writes new DDS samples on the same DataWriter, the new DDS samples must be acknowledged before unblocking the thread waiting on `wait_for_acknowledgments()`.

```cpp
DDS_ReturnCode_t wait_for_acknowledgments(
    const DDS_Duration_t & max_wait)
```

This operation returns DDS_RETCODE_OK if all the DDS samples were acknowledged, or DDS_RETCODE_TIMEOUT if the max_wait duration expired first.

If the DataWriter does not have its 7.5.21 RELIABILITY QosPolicy on page 447 kind set to RELIABLE, the operation will immediately return DDS_RETCODE_OK.

There is a similar operation available at the Publisher level, see 7.2.7 Waiting for Acknowledgments in a Publisher on page 286.

The reliability protocol used by Connext DDS is discussed in Reliable Communications (Chapter 11 on page 688). The application acknowledgment mechanism is discussed in 7.3.12 Application Acknowledgment below and Guaranteed Delivery of Data (Chapter 14 on page 752).

### 7.3.12 Application Acknowledgment

The 7.5.21 RELIABILITY QosPolicy on page 447 determines whether or not data published by a DataWriter will be reliably delivered by Connext DDS to matching DataReaders. The reliability protocol used by Connext DDS is discussed in Reliable Communications (Chapter 11 on page 688).

With protocol-level reliability alone, the producing application knows that the information is received by the protocol layer on the consuming side. However, the producing application cannot be certain that the consuming application read that information or was able to successfully understand and process it. The information could arrive in the consumer’s protocol stack and be placed in the DataReader cache but the consuming application could either crash before it reads it from the cache, not read its cache, or read the cache using queries or conditions that prevent that particular DDS data sample from being accessed. Fur-
therefore, the consuming application could access the DDS sample, but not be able to interpret its meaning or process it in the intended way.

The mechanism to let a DataWriter know to keep the DDS sample around, not just until it has been acknowledged by the reliability protocol, but until the application has been able to process the DDS sample is aptly called Application Acknowledgment. A reliable DataWriter will keep the DDS samples until the application acknowledges the DDS samples. When the subscriber application is restarted, the middleware will know that the application did not acknowledge successfully processing the DDS samples and will resend them.

7.3.12.1 Application Acknowledgment Kinds

Connext DDS supports three kinds of application acknowledgment, which is configured in the 7.5.21 RELIABILITY QosPolicy on page 447):

1. DDS_PROTOCOL_ACKNOWLEDGMENT_MODE (Default): In essence, this mode is identical to using no application-level acknowledgment. DDS samples are acknowledged according to the Real-Time Publish-Subscribe (RTPS) reliability protocol. RTPS AckNack messages will acknowledge that the middleware received the DDS sample.

2. DDS_APPLICATION_AUTO_ACKNOWLEDGMENT_MODE: DDS samples are automatically acknowledged by the middleware after the subscribing application accesses them, either through calling take() or read() on the DDS sample. If the read() or take() operation loans the samples, the acknowledgment is done after the return_loan() operation is called. Otherwise, for read() or take() operations that make a copy, acknowledgment is done after the read() or take() operations are executed.

3. DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE: DDS samples are acknowledged after the subscribing application explicitly calls acknowledge on the DDS sample. This can be done by either calling the DataReader’s acknowledge_sample() or acknowledge_all() operations. When using acknowledge_sample(), the application will provide the DDS_SampleInfo to identify the DDS sample being acknowledge. When using acknowledge_all, all the DDS samples that have been read or taken by the reader will be acknowledged.

Note: Even in DDS_APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE, some DDS samples may be automatically acknowledged. This is the case when DDS samples are filtered out by the reader using time-based filter, or using content filters. Additionally, when the reader is explicitly configured to use KEEP_LAST history kind, DDS samples may be replaced in the reader queue due to resource constraints. In that case, the DDS sample will be automatically acknowledged by the middleware if it has not been read by the application before it was replaced. To truly guarantee successful processing of DDS samples, it is recommended to use KEEP_ALL history kind.
7.3.12.2 Explicitly Acknowledging a Single DDS Sample (C++)

```cpp
void MyReaderListener::on_data_available(DDSDataReader *reader) {
    Foo sample;
    DDS_SampleInfo info;
    FooDataReader* fooReader = FooDataReader::narrow(reader);
    DDS_ReturnCode_t retcode = fooReader->take_next_sample(sample, info);
    if (retcode == DDS_RETCODE_OK) {
        if (info.valid_data) {
            // Process sample
            ...
            retcode = reader->acknowledge_sample(info);
            if (retcode != DDS_RETCODE_OK) {
                // Error
            }
        } else {
            // Not OK or NO DATA
        }
    } else {
    }
}
```

7.3.12.3 Explicitly Acknowledging All DDS samples (C++)

```cpp
void MyReaderListener::on_data_available(DDSDataReader *reader) {
    ...
    // Loop while samples available
    for(;;) {
        retcode = string_reader->take_next_sample(sample, info);
        if (retcode == DDS_RETCODE_NO_DATA) {
            // No more samples
            break;
        }
        // Process sample
        ...
    }
    retcode = reader->acknowledge_all();
    if (retcode != DDS_RETCODE_OK) {
        // Error
    }
}
```

7.3.12.4 Notification of Delivery with Application Acknowledgment

A `DataWriter` can get notification of delivery with Application Acknowledgment using two different mechanisms:

- `DataWriter's` `wait_for_acknowledgments()` operation
A **DataWriter** can use the `wait_for_acknowledgments()` operation to be notified when all the DDS samples in the **DataWriter**’s queue have been acknowledged. See 7.3.11 Waiting for Acknowledgments in a DataWriter on page 317.

```c
retCode = fooWriter->write(sample, DDS_HANDLE_NIL);
if (retCode != DDS_RETCODE_OK) {
    // Error
}
retcode = writer->wait_for_acknowledgments(timeout);
if (retCode != DDS_RETCODE_OK) {
    if (retCode == DDS_RETCODE_TIMEOUT) { // Timeout: Sample not acknowledged yet
        // Error
    } else {
        // Error
    }
}
```

Using `wait_for_acknowledgments()` does not provide a way to get delivery notifications on a per **DataReader** and DDS sample basis. If your application requires acknowledgment of message receipt, use the the second mechanism described below.

- **DataWriter's** listener callback **on_application_acknowledgment()**

An application can install a **DataWriter** listener callback **on_application_acknowledgment()** to receive a notification when a DDS sample is acknowledged by a **DataReader**. As part of this notification, you can access:

- The subscription handle of the acknowledging **DataReader**.
- The Identity of the DDS sample being acknowledged.
- The response data associated with the DDS sample being acknowledged.

For more information, see 7.3.6.1 APPLICATION_ACKNOWLEDGMENT_STATUS on page 296.

### 7.3.12.5 Application-Level Acknowledgment Protocol

When the subscribing application confirms it has successfully processed a DDS sample, an AppAck RTPS message is sent to the publishing application. This message will be resent until the publishing application confirms receipt of the AppAck message by sending an AppAckConf RTPS message. See Figure 7.10: AppAck RTPS Messages Sent when Application Acknowledges a DDS Sample on the next page through Figure 7.12: AppAck RTPS Messages Sent as a Sequence of Intervals, Combined to Optimize for Bandwidth on page 322.
Figure 7.10: AppAck RTPS Messages Sent when Application Acknowledges a DDS Sample

Figure 7.11: AppAck RTPS Messages Resent Until Acknowledged Through AppAckConf RTPS Message
Figure 7.12: AppAck RTPS Messages Sent as a Sequence of Intervals, Combined to Optimize for Bandwidth

7.3.12.6 Periodic and Non-Periodic AppAck Messages

You can configure whether AppAck RTPS messages are sent immediately or periodically through the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570. The samples_per_app_ack on page 574 (in Table 8.19 DDS_RtpsReliableReaderProtocol_t) determines the minimum number of DDS samples acknowledged by one application-level Acknowledgment message. The middleware will not send an AppAck message until it has at least this many DDS samples pending acknowledgment. By default, samples_per_app_ack is 1 and the AppAck RTPS message is sent immediately. Independently, the app_ack_period on page 573 (in Table 8.19 DDS_RtpsReliableReaderProtocol_t) determines the rate at which a DataReader will send AppAck messages.

7.3.12.7 Application Acknowledgment and Persistence Service

Application Acknowledgment is fully supported by RTI Persistence Service. The combination of Application Acknowledgment and Persistence Service is actually a common configuration. In addition to keeping DDS samples available until fully acknowledged, Persistence Service, when used in peer-to-peer mode, can take advantage of AppAck messages to avoid sending duplicate messages to the subscribing application. Because AppAck messages are sent to all matching writers, when the subscriber acknowledges the original publisher, Persistence Service will also be notified of this event and will not send out duplicate messages. This is illustrated in Figure 7.13: Application Acknowledgment and Persistence Service on the next page.
7.3.12.8 Application Acknowledgment and Routing Service

Application Acknowledgment is supported by RTI Routing Service: That is, Routing Service will acknowledge the DDS sample it has processed. Routing Service is an active participant in the Connext DDS system and not transparent to the publisher or subscriber. As such, Routing Service will acknowledge to the publisher, and the subscriber will acknowledge to Routing Service. However, the publisher will not get a notification from the subscriber directly.

7.3.13 Required Subscriptions

The 7.5.9 DURABILITY QosPolicy on page 411 specifies whether acknowledged DDS samples need to be kept in the DataWriter’s queue and made available to late-joining applications. When a late joining application is discovered, available DDS samples will be sent to the late joiner. With the Durability QoS alone, there is no way to specify or characterize the intended consumers of the information and you do not have control over which DDS samples will be kept for late-joining applications. If while waiting for late-joining applications, the middleware needs to free up DDS samples, it will reclaim DDS samples if they have been previously acknowledged by active/matching readers.
There are scenarios where you know a priori that a particular set of applications will join the system: e.g., a logging service or a known processing application. The Required Subscription feature is designed to keep data until these known late joining applications acknowledge the data.

Another use case is when DataReaders become temporarily inactive due to not responding to heartbeats, or when the subscriber temporarily became disconnected and purged from the discovery database. In both cases, the DataWriter will no longer keep the DDS sample for this DataReader. The Required Subscription feature will keep the data until these known DataReaders have acknowledged the data.

To use Required Subscriptions, the DataReaders and Data Writers must have their 7.5.21 RELIABILITY QosPolicy on page 447 kind set to RELIABLE.

### 7.3.13.1 Named, Required and Durable Subscriptions

Before describing the Required Subscriptions, it is important to understand a few concepts:

- **Named Subscription**: Through the 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418, each DataReader can be given a specific name. This name can be used by tools to identify a specific DataReader. Additionally, the DataReader can be given a role_name. For example: LOG_APP_1 DataReader belongs to the logger applications (role_name = “LOGGER”).

- **Required Subscription** is a named subscription to which a DataWriter is configured to deliver data to. This is true even if the DataReaders serving those subscriptions are not available yet. The DataWriter must store the DDS sample until it has been acknowledged by all active reliable DataReaders and acknowledged by all required subscriptions. The DataWriter is not waiting for a specific DataReader, rather it is waiting for DataReaders belonging to the required subscription by setting their role_name to the subscription name.

- **Durable Subscription** is a required subscription where DDS samples are stored and forwarded by an external service. In this case, the required subscription is served by RTI Persistence Service. See 46.9 Configuring Durable Subscriptions in Persistence Service on page 1200.

### 7.3.13.2 Durability QoS and Required Subscriptions

The 7.5.9 DURABILITY QosPolicy on page 411 and the Required Subscriptions feature complement each other.

The DurabilityQosPolicy determines whether or not Connext DDS will store and deliver previously acknowledged DDS samples to new DataReaders that join the network later. You can specify to either not make the DDS samples available (DDS_VOLATILE_DURABILITY_QOS kind), or to make them available and declare you are storing the DDS samples in memory (DDS_TRANSIENT_LOCAL_DURABILITY_QOS or DDS_TRANSIENT_DURABILITY_QOS kind) or in permanent storage (DDS_PERSISTENT_DURABILITY_QOS).
Required subscriptions help answer the question of when a DDS sample is considered acknowledged before the DurabilityQosPolicy determines whether to keep it. When required subscriptions are used, a DDS sample is considered acknowledged by a DataWriter when both the active DataReaders and a quorum of required subscriptions have acknowledged the DDS sample. (Acknowledging a DDS sample can be done either at the protocol or application level—see 7.3.12 Application Acknowledgment on page 317).

### 7.3.13.3 Required Subscriptions Configuration

Each DataReader can be configured to be part of a named subscription, by giving it a role_name using the 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418. A DataWriter can then be configured using the 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370 (required_matched_endpoint_groups) with a list of required named subscriptions identified by the role_name. Additionally, the DataWriter can be configured with a quorum or minimum number of DataReaders from a given named subscription that must receive a DDS sample.

When configured with a list of required subscriptions, a DataWriter will store a DDS sample until the DDS sample is acknowledged by all active reliable DataReaders, as well as all required subscriptions. When a quorum is specified, a minimum number of DataReaders of the required subscription must acknowledge a DDS sample in order for the DDS sample to be considered acknowledged. Specifying a quorum provides a level of redundancy in the system as multiple applications or services acknowledge they have received the DDS sample. Each individual DataReader is identified using its own virtual GUID (see 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570).

### 7.3.14 Managing Instances (Working with Keyed Data Types)

This section applies only to data types that use keys; see 2.4 DDS Samples, Instances, and Keys on page 18. Using the following operations for non-keyed types has no effect. This section describes how instances work on DataReaders. See also Chapter 6 Working with Instances on page 257.

Topics come in two flavors: those whose associated data type has specified some fields as defining the ‘key,’ and those whose associated data type has not. An example of a data-type that specifies key fields is shown in Figure 7.14: Data Type with a Key below.

**Figure 7.14: Data Type with a Key**

```c
typedef struct Flight {
    @key int32 flightId;
    string departureAirport;
    string arrivalAirport;
    Time_t departureTime;
    Time_t estimatedArrivalTime;
    Location_t currentPosition;
} ;
```
7.3.14 Managing Instances (Working with Keyed Data Types)

7.3.14.1 Writing Instances

If the data type has some fields that act as a ‘key,’ the Topic contains one or more instances whose values can be independently maintained. In Figure 7.14: Data Type with a Key on the previous page, the flightId is the ‘key’. Different flights will have different values for the key. Each flight is an instance of the Topic. Each write() (or write() variation such as write_w_timestamp()) will update the information about a single flight—meaning that when a DataWriter calls write(), the DataWriter is updating the instance represented by the flightId.

When a DataWriter updates an instance by calling write(), a sample of that instance is sent to matching DataReaders, and the DataReaders consider the instance to be ALIVE.

If the DataWriter’s 7.5.22 RESOURCESLIMITS QosPolicy on page 451 specifies a max_instances limit that is not infinite, the limit will apply when writing. If a DataWriter writes an instance that it has not written before, and it has already reached the max_instances limit, it will try to reclaim the memory used by an existing instance. The rules for which instances it can replace are described in 7.3.14.7 Instance Memory Management on page 330.

If the DataWriter cannot reclaim the memory used by an existing instance, the write() call will fail. For more information on the behavior of the write() call when the max_instances limit is hit, see 7.3.8.2 write() behavior with KEEP_LAST and KEEP_ALL on page 312.

7.3.14.2 Registering Instances

If your data type has a key, you may improve performance of any operation that modifies the instance, such as any variation of write() or dispose(), by providing a non-NIL instance handle. An instance handle contains the pre-calculated instance keyhash so that it does not need to be calculated again. The instance handle for an instance can be retrieved once the instance is registered.

A DataWriter can register and retrieve an instance handle for an instance in two ways:

- Explicitly, with the register_instance() operation. The register_instance() operation provides a handle to the instance (of type DDS_InstanceHandle_t) that can be used later to refer to the instance.
- Implicitly by providing a NIL instance handle to one of the variations of the write() or dispose() calls. After one of these calls has been made, the instance handle for the now-registered instance can be retrieved using the DataWriter lookup_instance() call.

Once you have an instance handle, you can use it while writing to avoid calculating the instance keyhash in every write call. This performance improvement may be significant if your data is relatively small or your key fields are relatively complex. (If your data itself is large or complex, the time to calculate the keyhash may be insignificant relative to the time to serialize your data.)

You can register any number of instances up to the maximum number of instances configured in the DataWriter’s 7.5.22 RESOURCESLIMITS QosPolicy on page 451. Explicit instance registration is
7.3.14 Managing Instances (Working with Keyed Data Types)

When you keep DataWriter instances alive for longer, it completely optional. Note that registration through the `register_instance()` call only affects the `DataWriter`: matching DataReaders are not notified that the instance is ALIVE when the `DataWriter` registers the instance. An instance is only recognized as ALIVE when a `DataWriter` writes the instance. When an application registers instances and uses the instance handles for increased performance, it must keep a mapping between instance handles and instances. See the **Warning** below.

**Figure 7.15: Explicitly Registering an Instance**

```cpp
Flight myFlight;
// writer is a previously-created FlightDataWriter
myFlight.flightId = 265;
DDS_InstanceHandle_t f1265Handle = 
writer->register_instance(myFlight);
...
// Each time we update the flight, we can pass the handle
myFlight.departureAirport = "SJC";
myFlight.arrivalAirport = "LAX";
myFlight.departureTime = {120000, 0};
myFlight.arrivalAirport = {130200, 0};
myFlight.currentPosition = { {37, 20}, {121, 53} };
if (writer->write(myFlight, f1265Handle) != DDS_RETCODE_OK) {
  // ... handle error
}
// The writer can declare that it will no longer update information about
// this flight by unregistering itself from the instance
if (writer->unregister_instance(myFlight, f1265Handle) !=
  DDS_RETCODE_OK) {
  // ... handle error
}
```

**Warning:** If you decide to manage instance handles using your own application logic, make sure you keep a correct mapping between the instance and instance handle. If you pass the wrong instance handle when writing data, *Connext DDS* will assume that you are writing the instance associated with the handle. It does not check that the key fields match that handle, because that would negate the performance improvement from passing the handle. Passing the wrong instance handle can lead to strange behavior, because *Connext DDS* will treat your data sample as though it belongs to the wrong instance.

For example, if you have the History QosPolicy **kind** set to KEEP_LAST and **depth** set to 1 on the `DataReader`, *Connext DDS* should keep the last sample for each instance. But if you pass the wrong instance handle when writing, the `DataReader` will overwrite the wrong sample (in the wrong instance). As a result, a `DataReader` will not get updates for the instance it expects. An incorrect instance handle will affect all QoS policies that are applied per instance; see 6.3.1 QoS Policies that are Applied per Instance on page 266.

When you are done using an instance, you can unregister it. See **7.3.14.4 Unregistering Instances** below.
7.3.14.3 Disposing Instances

The `dispose()` operation informs `DataReaders` that, as far as the `DataWriter` knows, the instance no longer exists and can be considered “not alive.” When the `dispose()` operation is called, the instance state stored in the `DDS_SampleInfo` structure, accessed through `DataReaders`, will change to `NOT_ALIVE_DISPOSED` for that particular instance.

Often, systems use the `NOT_ALIVE_DISPOSED` state to indicate that some object is completely gone from the system. For example, in a flight tracking system, when a flight lands, a `DataWriter` may dispose of the instance corresponding to the flight. In that case, all `DataReaders` who are monitoring the flight will see the instance state change to `NOT_ALIVE_DISPOSED`, indicating that the flight has landed.

**Note:** If a `DataWriter` calls `dispose()`, it does not give up ownership of the instance (unlike when it calls `unregister_instance()`), in which case it is declaring that it will no longer have any updates for the instance and therefore does give up ownership of the instance to other `DataWriters` that may still be actively updating the instance.

**Attention:** Disposing does not free up memory by default. For instance, when the `DataWriter` calls `dispose()` to indicate that a flight has landed, it must keep the dispose message in its queue so all matching `DataReaders` get notified that the flight has landed (i.e., has been disposed). Also, in terms of memory management, *Connext DDS* may reclaim unregistered instances before disposed ones, or not reclaim disposed instances at all, depending on your QoS settings. See 7.3.14.7 Instance Memory Management on page 330.

See also:

- 7.5.31.1 Unregistering vs. Disposing on page 469
- 7.5.5.5 Propagating Serialized Keys with Disposed-Instance Notifications on page 398
- 7.5.31.2 Autodisposing Unregistered Instances on page 469

7.3.14.4 Unregistering Instances

The `unregister_instance()` operation informs `DataReaders` that the `DataWriter` is no longer updating the instance. When a `DataWriter` will no longer update an instance, you can unregister it. To unregister a `DataWriter` from an instance, use the `DataWriter`’s `unregister_instance()` operation. Unregistering tells *Connext DDS* that the `DataWriter` has no more information on this instance; thus, it does not intend to modify that instance anymore, allowing *Connext DDS* to recover any resources it allocated for the instance.

`unregister_instance()` should only be used on instances that have been previously registered. Instances can be registered explicitly with the `register_instance()` operation, or implicitly with any variation of the `write()` or `dispose()` operations. See Figure 7.15: Explicitly Registering an Instance on page 327.

Once all `DataWriters` have unregistered from an instance, the matched `DataReaders` will eventually get an indication that the instance no longer has any `DataWriters`. This is communicated to the subscribing
application by means of the `DDS_SampleInfo` that accompanies each DDS sample (see 8.4.6 The SampleInfo Structure on page 565). Once there are no `DataWriters` for the instance, the `DataReader` will see the value of `DDS_InstanceStateKind` for that instance to be `NOT_ALIVE_NO_WRITERS`.

Note that `DataReaders` can’t distinguish between a scenario where all `DataWriters` explicitly unregister from an instance and a scenario where all `DataWriters` have lost liveliness. For more information on `DataWriter` liveliness, see the 7.5.15 LIVELINESS QoS Policy on page 427.

The `unregister_instance()` operation may affect the ownership of the instance (see the 7.5.17 OWNERSHIP QoS Policy on page 434). If the `DataWriter` was the exclusive owner of the instance, then calling `unregister_instance()` relinquishes that ownership, and another `DataWriter` can become the exclusive owner of the instance. (In contrast, if a `DataWriter` calls `dispose()`, it does not give up ownership of the instance.)

The `unregister_instance()` operation indicates only that a particular `DataWriter` no longer has any information/data on an instance and thus no longer has anything to say about the instance. It does not indicate that anything about the instance itself has changed, such as its existence or the associated data. For example, a `DataWriter` that is tracking a flight may unregister from an instance when the flight goes out of range—this does not mean that the position of the flight has changed or that the flight has landed, just that the `DataWriter` no longer has any knowledge of the flight; other `DataWriters` may still update the flight’s position.

The `autodispose_unregistered_instances` field in the 7.5.31 WRITER_DATA_LIFECYCLE QoS Policy on page 467 controls whether instances are automatically disposed of when they are unregistered. (By default, they are not. See 7.5.31.2 Autodisposing Unregistered Instances on page 469.) When this QoS is true and the `DataWriter` unregisters from an instance, two samples are sent to the `DataReader` to notify it that the instance is both unregistered and disposed. The rules about which instance memory can be reclaimed are documented in 7.3.14.7 Instance Memory Management on the next page.

The `unregister_instance()` operation adds one sample (or two) to the `DataReader` queue, so the behavior of `unregister_instance()` with regards to `KEEP_LAST` and `KEEP_ALL` is the same as for the `write()` operation. See 7.3.8.2 `write()` behavior with `KEEP_LAST` and `KEEP_ALL` on page 312. (Two samples are added if `autodispose_unregistered_instances` is set to `TRUE`; `Connext DDS` makes a dispose and an unregister sample. See `autodispose_unregistered_instances` on page 468 in the 7.5.31 WRITER_DATA_LIFECYCLE QoS Policy on page 467.)

See also:

- 7.5.31.1 Unregistering vs. Disposing on page 469
- 7.5.31.2 Autodisposing Unregistered Instances on page 469

7.3.14.5 Looking up an Instance Handle

Some operations, such as `write()`, accept an `instance_handle` parameter. If you need to get such a handle, you can call the `FooDataWriter`’s `lookup_instance()` operation, which takes an instance as a parameter.
and returns a handle to that instance. This is useful only for keyed data types.

\[
{\text{DDS\_InstanceHandle\_t\ probe_instance (const Foo & key\_holder)}}
\]

The instance must have already been registered, written, or disposed. If the instance is not known to the DataWriter, this operation returns DDS\_HANDLE\_NIL.

### 7.3.14.6 Getting the Key Value for an Instance

Once you have an instance handle (using register\_instance() or lookup\_instance()), you can use the DataWriter’s get\_key\_value() operation to retrieve the value of the key of the corresponding instance. The key fields of the data structure passed into get\_key\_value() will be filled out with the original values used to generate the instance handle. The key fields are defined when the data type is defined (see 2.4 DDS Samples, Instances, and Keys on page 18).

Following our example in Figure 7.15: Explicitly Registering an Instance on page 327, register\_instance () returns a DDS\_InstanceHandle\_t that can be used in the call to the FlightDataWriter’s get\_key\_value() operation. The value of the key is returned in a structure of type Flight with the flightId field filled in with the integer 265.

See also: 7.5.5.5 Propagating Serialized Keys with Disposed-Instance Notifications on page 398.

### 7.3.14.7 Instance Memory Management

In Connext DDS, memory is primarily pre-allocated when creating entities. When data is keyed, the memory associated with each instance used for storing instance-specific metadata is allocated when the DataWriter is created. Memory is not freed at runtime, unless you delete an entity. Instead, memory is made available to be reused by the DataWriter, or 'reclaimed'.

Instance memory in the DataWriter is reclaimed two ways:

- **Lazily (Default):** when a resource limit such as max\_instances is hit. Only once this limit is hit will Connext DDS reclaim memory as described in the following sections.
- **Proactively (Non-Default):** after a time delay, configured by autopurge\_unregistered\_instance\_delay or autopurge\_disposed\_instances\_delay, as long as all samples of that instance are fully-acknowledged (see 7.3.8.2 write() behavior with KEEP\_LAST and KEEP\_ALL on page 312). In this case, the instance data is purged, freeing up memory for future use (i.e., for "reclaiming").

In the default case, Connext DDS has to decide which instances to replace first. This is controlled by the following QoS policies and settings.

#### 7.3.14.7.1 WriterDataLifecycle: autopurge\_unregistered\_instances\_delay

When autopurge\_unregistered\_instances\_delay in the 7.5.31 WRITER\_DATA\_LIFECYCLE QoS Policy on page 467 is 0, Connext DDS will clean up all the resources associated with an unregistered instance (most notably, the DDS sample history of non-volatile DataWriters) when all the instance’s
samples have been acknowledged by all its live DataReaders, including the sample that indicates the unregistration. By default, autopurge_unregistered_instances_delay is disabled (the delay is INFINITE). If the delay is set to zero, the DataWriter will clean up as soon as all the samples are acknowledged after the call to unregister_instance(). A non-zero value for the delay can be useful in two ways:

- To keep the historical DDS samples for late-joiners for a period of time.
- In the context of the builtin discovery DataWriters, if the applications temporarily lose the connection before the unregistration (which represents the remote entity destruction), to provide the DDS samples that indicate the dispose and unregister actions once the connection is reestablished.

This delay can also be set for discovery data through these fields in the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644:

- publication_writer_data_lifecycle.autopurge_unregistered_instances_delay
- subscription_writer_data_lifecycle.autopurge_unregistered_instances_delay
- publication_writer_data_lifecycle.autopurge_disposed_instances_delay
- subscription_writer_data_lifecycle.autopurge_disposed_instances_delay

7.3.14.7.2 DataWriterResourceLimits: replace_empty_instances

The replace_empty_instances field in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401 defines whether instances with no samples in the DataWriter queue be replaced first, regardless of their instance state. If there are multiple empty instances, replace_empty_instances will replace unregistered instances, then disposed instances, then alive instances. If replace_empty_instances is true, empty instances will always be replaced first before any instance that may qualify for replacement based on the instance_replacement field in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401.

Values: true/false

7.3.14.7.3 DataWriterResourceLimits: instance_replacement

This instance_replacement field in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401 defines which instance states can be replaced, and the order in which they are allowed to be replaced. This setting takes effect if all samples for an instance are fully acknowledged.

Values:

- DDS_UNREGISTERED_INSTANCE_REPLACEMENT
- DDS_ALIVE_INSTANCE_REPLACEMENT
- DDS_DISPOSED_INSTANCE_REPLACEMENT
7.3.14 Managing Instances (Working with Keyed Data Types)

- DDS_ALIVE_THEN_DISPOSED_INSTANCE_REPLACEMENT
- DDS_DISPOSED_THEN_ALIVE_INSTANCE_REPLACEMENT
- DDS_ALIVE_OR_DISPOSED_INSTANCE_REPLACEMENT

**Warning:** Unregistered instances are always replaced first even if you don't choose DDS_UNREGISTERED_INSTANCE_REPLACEMENT. Because unregistering an instance indicates that the DataWriter will no longer update the instance, it is assumed that reclaiming these resources first will avoid information loss in your system.

When a DataWriter disposes an instance, it cannot replace the memory related to that instance unless autopurge_disposed_instances_delay is finite, the instance replacement field in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401 indicates that disposed instances can be replaced when instance resource limits are reached, or the instance is empty and replace_empty_instances is true.

See also:
- 7.5.31.1 Unregistering vs. Disposing on page 469
- 7.5.31.2 Autodisposing Unregistered Instances on page 469

### 7.3.14.8 Consequences of Unpurged Dispose Messages

There are consequences of having many unpurged dispose messages in the DataWriter's queue. If the DataWriter’s 7.5.9 DURABILITY QosPolicy on page 411 kind is not VOLATILE, those dispose messages will be delivered to late-joining DataReaders, which may cause an unexpected spike in network traffic. In addition, the DataReaders will not notify the application about those previously-disposed instances, because by default DataReaders will not propagate dispose messages for instances that were previously unknown. (This behavior can be changed by using the `propagate_dispose_of_unregistered_instances` QoS setting on the DataReader.)

Failing to purge disposed instances will cause similar behavior when using TopicQueries. When the DataWriter sends the response to the TopicQuery, it will include the unpurged dispose messages, causing high network traffic. In general, all dispose and unregister messages always pass filters (associated with ContentFilteredTopics, TopicQueries, or QueryConditions). This means that even if a TopicQuery’s filter expression only specifies a specific key value or set of key values, all dispose messages for all instances in the DataWriter queue will be sent in response to the TopicQuery. To avoid this when using TopicQueries, use the special statement at the beginning of the query: “@instance_state = ALIVE AND” followed by the rest of the expression. This prevents the DataWriter from sending not-alive samples.

See also:
7.3.15 Setting DataWriter QosPolicies

- 7.5.5.5 Propagating Serialized Keys with Disposed-Instance Notifications on page 398
- 7.5.31.2 Autodisposing Unregistered Instances on page 469

7.3.14.9 Consequences of DataWriters Reclaiming Disposed Instances

If your network is subject to disconnections, and disposed instances are purged, it’s possible that a dispose message is not received by every DataReader, leading to DataReaders recognizing different instance states. This happens if your network disconnection is long enough for a reliable DataReader to be marked as inactive and the disposed instance is purged during the disconnection. If the disposed message is not purged during the disconnection, it is still possible for the dispose message to be delivered after reconnection if the 7.5.9 DURABILITY QosPolicy on page 411 is not VOLATILE.

If you have one or more RTI Routing Service applications in your network, leading to multiple places where instance state gets cached and might be reclaimed, it is even more likely that a dispose message might not be received by every DataReader.

7.3.15 Setting DataWriter QosPolicies

The DataWriter’s QosPolicies control its resources and behavior.

The DDS_DataWriterQos structure has the following format:

```c
DDS_DataWriterQos struct {
  DDS_DurabilityQosPolicy durability;
  DDS_DurabilityServiceQosPolicy durability_service;
  DDS_DeadlineQosPolicy deadline;
  DDS_LatencyBudgetQosPolicy latency_budget;
  DDS_LivelinessQosPolicy liveliness;
  DDS_ReliabilityQosPolicy reliability;
  DDS_DestinationOrderQosPolicy destination_order;
  DDS_HistoryQosPolicy history;
  DDS_ResourceLimitsQosPolicy resource_limits;
  DDS_TransportPriorityQosPolicy transport_priority;
  DDS_LifespanQosPolicy lifespan;
  DDS_UserDataQosPolicy user_data;
  DDS_OwnershipQosPolicy ownership;
  DDS_OwnershipStrengthQosPolicy ownership_strength;
  DDS_WriterDataLifecycleQosPolicy writer_data_lifecycle;
  DDS_DataRepresentationQosPolicy representation;
  DDS_DataTagQosPolicy data_tags;
  // extensions to the DDS standard:
  DDS_DataWriterResourceLimitsQosPolicy writer_resource_limits;
  DDS_DataWriterProtocolQosPolicy protocol;
  DDS_TransportSelectionQosPolicy transport_selection;
  DDS_TransportUnicastQosPolicy unicast;
  DDS_PublishModeQosPolicy publish_mode;
  DDS_PropertyQosPolicy property;
  DDS_ServiceQosPolicy service;
};
```
Note: set_qos() cannot always be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

Table 7.18 DataWriter QosPolicies summarizes the meaning of each policy. (They appear alphabetically in the table.) For information on why you would want to change a particular QosPolicy, see the referenced section. For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 7.18 DataWriter QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
</table>
| Availability                | This QoS policy is used in the context of two features:  
7.5.1.1 Availability Qos Policy and Collaborative DataWriters on page 371  
7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370  
For Collaborative DataWriters, Availability specifies the group of DataWriters expected to collaboratively provide data and the timeouts that control when to allow data to be available that may skip DDS samples.  
For Required Subscriptions, Availability configures a set of Required Subscriptions on a DataWriter.  
See 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370 |
| Batch                       | Specifies and configures the mechanism that allows Connext DDS to collect multiple DDS user data samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput. See 7.5.2 BATCH QosPolicy (DDS Extension) on page 374. |
| DataRepresentation          | Specifies which version of the Extended Common Data Representation (CDR) is offered. See 7.5.3 DATA_REPRESENTATION QosPolicy on page 380.                                                                 |
| DataTag                     | A sequence of (name, value) string pairs that may be used by the Access Control plugin. See 7.5.4 DATATAG QosPolicy on page 388.                                                                 |
| DataWriterProtocol          | This QosPolicy configures the Connext DDS on-the-network protocol, RTPS. See 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.                                                             |
| DataWriterResourceLimits    | Controls how many threads can concurrently block on a write() call of this DataWriter. See 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 401.                                    |
| Deadline                    | For a DataReader, it specifies the maximum expected elapsed time between arriving DDS data samples.  
For a DataWriter, it specifies a commitment to publish DDS samples with no greater elapsed time between them.  
See 7.5.7 DEADLINE QosPolicy on page 405.                                                                              |
| DestinationOrder            | Controls how Connext DDS will deal with data sent by multiple DataWriters for the same topic. Can be set to "by reception timestamp" or to "by source timestamp". See 7.5.8 DESTINATION_ORDER QosPolicy on page 408. |
### Table 7.18 DataWriter QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Durability</td>
<td>Specifies whether or not Connext DDS will store and deliver data that were previously published to new DataReaders. See 7.5.9 DURABILITY QosPolicy on page 411.</td>
</tr>
<tr>
<td>DurabilityService</td>
<td>Various settings to configure the external Persistence Service used by Connext DDS for DataWriters with a Durability QoS setting of Persistent Durability. See 7.5.10 DURABILITY SERVICE QosPolicy on page 416.</td>
</tr>
<tr>
<td>EntityName</td>
<td>Assigns a name to a DataWriter. See 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
<tr>
<td>History</td>
<td>Specifies how much data must to be stored by Connext DDS for the DataWriter or DataReader. This QosPolicy affects the 7.5.21 RELIABILITY QosPolicy on page 447 as well as the 7.5.9 DURABILITY QosPolicy on page 411. See 7.5.12 HISTORY QosPolicy on page 420.</td>
</tr>
<tr>
<td>LatencyBudget</td>
<td>Suggestion to Connext DDS on how much time is allowed to deliver data. See 7.5.13 LATENCYBUDGET QoS Policy on page 425.</td>
</tr>
<tr>
<td>Lifespan</td>
<td>Specifies how long Connext DDS should consider data sent by an user application to be valid. See 7.5.14 LIFESPAN Qos Policy on page 425.</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Specifies and configures the mechanism that allows DataReaders to detect when DataWriters become disconnected or &quot;dead.&quot; See 7.5.15 LIVELINESS QosPolicy on page 427.</td>
</tr>
<tr>
<td>MultiChannel</td>
<td>Configures a DataWriter’s ability to send data on different multicast groups (addresses) based on the value of the data. See 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432.</td>
</tr>
<tr>
<td>Ownership</td>
<td>Along with OwnershipStrength, specifies if DataReaders for a topic can receive data from multiple DataWriters at the same time. See 7.5.17 OWNERSHIP QosPolicy on page 434.</td>
</tr>
<tr>
<td>OwnershipStrength</td>
<td>Used to arbitrate among multiple DataWriters of the same instance of a Topic when Ownership QosPolicy is EXCLUSIVE. See 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438.</td>
</tr>
<tr>
<td>Partition</td>
<td>Adds string identifiers that are used for matching DataReaders and DataWriters for the same Topic. See 7.5.5 PARTITION QosPolicy on page 356.</td>
</tr>
<tr>
<td>Property</td>
<td>Stores name/value (string) pairs that can be used to configure certain parameters of Connext DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.</td>
</tr>
<tr>
<td>PublishMode</td>
<td>Specifies how Connext DDS sends application data on the network. By default, data is sent in the user thread that calls the DataWriter’s write() operation. However, this QosPolicy can be used to tell Connext DDS to use its own thread to send the data. See 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Specifies whether or not Connext DDS will deliver data reliably. See 7.5.21 RELIABILITY QosPolicy on page 447.</td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>Controls the amount of physical memory allocated for Entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.</td>
</tr>
<tr>
<td>Service</td>
<td>Intended for use by RTI infrastructure services. User applications should not modify its value. See 7.5.23 SERVICE QosPolicy (DDS Extension) on page 454.</td>
</tr>
</tbody>
</table>
Table 7.18 DataWriter QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TopicQueryDispatch</td>
<td>Configures the ability of a DataWriter to publish samples in response to a TopicQuery. See 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455.</td>
</tr>
<tr>
<td>TransferMode</td>
<td>Configures the properties of a Zero Copy DataWriter. See 7.5.25 TRANSFER_MODE QosPolicy on page 457.</td>
</tr>
<tr>
<td>TransportPriority</td>
<td>Set by a DataWriter to tell Connext DDS that the data being sent is a different “priority” than other data. See 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458.</td>
</tr>
<tr>
<td>TransportSelection</td>
<td>Allows you to select which physical transports a DataWriter or DataReader may use to send or receive its data. See 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459.</td>
</tr>
<tr>
<td>TransportUnicast</td>
<td>Specifies a subset of transports and port number that can be used by an Entity to receive data. See 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460.</td>
</tr>
<tr>
<td>TypeSupport</td>
<td>Used to attach application-specific value(s) to a DataWriter or DataReader. These values are passed to the serialization or deserialization routine of the associated data type. Also controls whether padding bytes are set to 0 during serialization. See 7.5.29 TYPESUPPORT QosPolicy (DDS Extension) on page 464.</td>
</tr>
<tr>
<td>UserData</td>
<td>Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to Connext DDS's discovery meta-data. See 7.5.30 USER_DATA QosPolicy on page 465.</td>
</tr>
<tr>
<td>WriterDataLifeCycle</td>
<td>Controls how a DataWriter handles the lifecycle of the instances (keys) that the DataWriter is registered to manage. See 7.5.31 WRITER_DATA_LIFECYCLE QosPolicy on page 467.</td>
</tr>
</tbody>
</table>

Many of the DataWriter QosPolicies also apply to DataReaders (see 8.3 DataReaders on page 504). For a DataWriter to communicate with a DataReader, their QosPolicies must be compatible. Generally, for the QosPolicies that apply both to the DataWriter and the DataReader, the setting in the DataWriter is considered an “offer” and the setting in the DataReader is a “request.” Compatibility means that what is offered by the DataWriter equals or surpasses what is requested by the DataReader. Each policy’s description includes compatibility restrictions. For more information on compatibility, see 4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property on page 180.

Some of the policies may be changed after the DataWriter has been created. This allows the application to modify the behavior of the DataWriter while it is in use. To modify the QoS of an already-created DataWriter, use the get_qos() and set_qos() operations on the DataWriter. This is a general pattern for all Entities, described in 4.1.7.3 Changing the QoS for an Existing Entity on page 175.

7.3.15.1 Configuring QoS Settings when the DataWriter is Created

As described in 7.3.1 Creating DataWriters on page 292, there are different ways to create a DataWriter, depending on how you want to specify its QoS (with or without a QoS Profile).

- In Figure 7.9: Creating a DataWriter with Default QosPolicies and a Listener on page 293, there is an example of how to create a DataWriter with default QosPolicies by using the special constant, DDS_DATAWRITER_QOS_DEFAULT, which indicates that the default QoS values for a
DataWriter should be used. The default DataWriter QoS values are configured in the Publisher or DomainParticipant; you can change them with set_default_datawriter_qos() or set_default_datawriter_qos_with_profile(). Then any DataWriters created with the Publisher will use the new default values. As described in 4.1.7 Getting, Setting, and Comparing QoS Policies on page 172, this is a general pattern that applies to the construction of all Entities.

- To create a DataWriter with non-default QoS without using a QoS Profile, see the example code in Figure 7.16: Creating a DataWriter with Modified QoS Policies (not from a profile) below. It uses the Publisher’s get_default_writer_qos() method to initialize a DDS_DataWriterQos structure. Then the policies are modified from their default values before the structure is used in the create_datawriter() method.

- You can also create a DataWriter and specify its QoS settings via a QoS Profile. To do so, you will call create_datawriter_with_profile(), as seen in Figure 7.17: Creating a DataWriter with a QoS Profile on the next page.

- If you want to use a QoS profile, but then make some changes to the QoS before creating the DataWriter, call get_datawriter_qos_from_profile() and create_datawriter() as seen in Figure 7.18: Getting QoS Values from a Profile, Changing QoS Values, Creating a DataWriter with Modified QoS Values on the next page.

For more information, see 7.3.1 Creating DataWriters on page 292 and Configuring QoS with XML (Chapter 19 on page 848).

Notes:

- The examples in this section use the Traditional C++ API; for examples in the Modern C++ API, see the sections "DataWriter Use Cases," "Qos Use Cases," and "Qos Provider Use Cases" in the API Reference HTML documentation, under "Programming How-To's."

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

Figure 7.16: Creating a DataWriter with Modified QoS Policies (not from a profile)

```cpp
DDS_DataWriterQos writer_qos;
// initialize writer_qos with default values
publisher->get_default_datawriter_qos(writer_qos);
// make QoS changes
writer_qos.history.depth = 5;
// Create the writer with modified qos
DDSDaataWriter * writer = publisher->create_datawriter(
    topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);
if (writer == NULL) {
    // ... error
}
// narrow it for your specific data type
FooDataWriter* foo_writer = FooDataWriter::narrow(writer);
```
7.3.15 Setting DataWriter QosPolicies

Figure 7.17: Creating a DataWriter with a QoS Profile

```c
// Create the datawriter
DDSDataWriter * writer =
    publisher->create_datawriter_with_profile(
        topic, "MyWriterLibrary", "MyWriterProfile",
        NULL, DDS_STATUS_MASK_NONE);
if (writer == NULL) {
    // ... error
};
// narrow it for your specific data type
FooDataWriter* foo_writer = FooDataWriter::narrow(writer);
```

Figure 7.18: Getting QoS Values from a Profile, Changing QoS Values, Creating a DataWriter with Modified QoS Values

```c
DDS_DataWriterQos writer_qos;
// Get writer QoS from profile
retcode = factory->get_datawriter_qos_from_profile(
    writer_qos, "WriterProfileLibrary", "WriterProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes
writer_qos.history.depth = 5;
DDSDataWriter * writer = publisher->create_datawriter(
    topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);
if (participant == NULL) {  
    // handle error
}
```

7.3.15.2 Comparing QoS Values

The `equals()` operation compares two `DataWriter`'s DDS_DataWriterQos structures for equality. It takes two parameters for the two `DataWriter`'s QoS structures to be compared, then returns TRUE if they are equal (all values are the same) or FALSE if they are not equal.

7.3.15.3 Changing QoS Settings After the DataWriter Has Been Created

There are two ways to change an existing `DataWriter`'s QoS after it is has been created—again depending on whether or not you are using a QoS Profile.

- To change QoS programmatically (that is, without using a QoS Profile), use `get_qos()` and `set_qos()` . See the example code in Figure 7.19: Changing the QoS of an Existing DataWriter (without a QoS Profile) below. It retrieves the current values by calling the `DataWriter`'s `get_qos()` operation. Then it modifies the value and calls `set_qos()` to apply the new value. Note, however, that some QosPolicies cannot be changed after the `DataWriter` has been enabled—this restriction is noted in the descriptions of the individual QosPolicies.
You can also change a DataWriter’s (and all other Entities’) QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see Figure 7.20: Changing the QoS of an Existing DataWriter with a QoS Profile below. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

Figure 7.19: Changing the QoS of an Existing DataWriter (without a QoS Profile)

```c
DDS_DataWriterQos writer_qos;
// Get current QoS.
if (datawriter->get_qos(writer_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
writer_qos.history.depth = 5;
// Set the new QoS
if (datawriter->set_qos(writer_qos) != DDS_RETCODE_OK) {
    // handle error
}
```

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

Figure 7.20: Changing the QoS of an Existing DataWriter with a QoS Profile

```c
retcode = writer->set_qos_with_profile(
    "WriterProfileLibrary","WriterProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

7.3.15.4 Using a Topic’s QoS to Initialize a DataWriter’s QoS

Several DataWriter QosPolicies can also be found in the QosPolicies for Topics (see 5.1.3 Setting Topic QosPolicies on page 219). The QosPolicies set in the Topic do not directly affect the DataWriters (or DataReaders) that use that Topic. In many ways, some QosPolicies are a Topic-level concept, even though the DDS standard allows you to set different values for those policies for different DataWriters and DataReaders of the same Topic. Thus, the policies in the DDS_TopicQos structure exist as a way to help centralize and annotate the intended or suggested values of those QosPolicies. Connext DDS does not check to see if the actual policies set for a DataWriter is aligned with those set in the Topic to which it is bound.

There are many ways to use the QosPolicies’ values set in the Topic when setting the QosPolicies’ values in a DataWriter. The most straightforward way is to get the values of policies directly from the Topic and use them in the policies for the DataWriter, as shown in Figure 7.21: Copying Selected QoS from a Topic when Creating a DataWriter below.
7.3.15 Setting DataWriter QosPolicies

Figure 7.21: Copying Selected QoS from a Topic when Creating a DataWriter

```c
DDS_DataWriterQos writer_qos;
DDS_TopicQos topic_qos;
// topic and publisher already created
// get current QoS for the topic, default QoS for the writer
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
if (publisher->get_default_datawriter_qos(writer_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Copy specific policies from topic QoS to writer QoS
writer_qos.deadline = topic_qos.deadline;
writer_qos.reliability = topic_qos.reliability;
// Create the DataWriter with the modified QoS
DDSDataWriter* writer = publisher->create_datawriter(topic,
    writer_qos, NULL, DDS_STATUS_MASK_NONE);
```

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

You can use the Publisher’s `copy_from_topic_qos()` operation to copy all of the common policies from the Topic QoS to a DataWriter QoS. This is illustrated in Figure 7.22: Copying all QoS from a Topic when Creating a DataWriter below.

Figure 7.22: Copying all QoS from a Topic when Creating a DataWriter

```c
DDS_DataWriterQos writer_qos;
DDS_TopicQos topic_qos;
// topic, publisher, writer_listener already created
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
if (publisher->get_default_datawriter_qos(writer_qos) != DDS_RETCODE_OK) {
    // handle error
}
// copy relevant QoS from topic into writer’s qos
publisher->copy_from_topic_qos(writer_qos, topic_qos);
// Optionally, modify policies as desired
writer_qos.deadline.duration.sec = 1;
writer_qos.deadline.duration.nanosec = 0;
// Create the DataWriter with the modified QoS
DDSDataWriter* writer = publisher->create_datawriter(topic,
    writer_qos, writer_listener, DDS_STATUS_MASK_ALL);
```
In another design pattern, you may want to start with the default QoS values for a DataWriter and override them with the QoS values of the Topic. Figure 7.23: Combining Default Topic and DataWriter QoS (Option 1) below gives an example of how to do this.

Because this is a common pattern, Connext DDS provides a special macro, DDS_DATAWRITER_QOS_USE_TOPIC_QOS, that can be used to indicate that the DataWriter should be created with the set of QoS values that results from modifying the default DataWriter QosPolicies with the QoS values specified by the Topic. Figure 7.24: Combining Default Topic and DataWriter QoS (Option 2) on the next page shows how the macro is used.

The code fragments shown in Figure 7.23: Combining Default Topic and DataWriter QoS (Option 1) below and Figure 7.24: Combining Default Topic and DataWriter QoS (Option 2) on the next page result in identical QoS settings for the created DataWriter.

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

Figure 7.23: Combining Default Topic and DataWriter QoS (Option 1)

```c
DDSDataWriterQos writer_qos;
DDS_TopicQos topic_qos;
// topic, publisher, writer_listener already created
if (topic-&gt;get_qos(topic_qos) != DDS_RETCODE_OK) {
  // handle error
}
if (publisher-&gt;get_default_datawriter_qos(writer_qos)
    != DDS_RETCODE_OK) {
  // handle error
}
if (publisher-&gt;copy_from_topic_qos(writer_qos, topic_qos)
    != DDS_RETCODE_OK) {
  // handle error
}
// Create the DataWriter with the combined QoS
DDSDataWriter* writer =
    publisher-&gt;create_datawriter(topic, writer_qos,
        writer_listener, DDS_STATUS_MASK_ALL);
```

Figure 7.24: Combining Default Topic and DataWriter QoS (Option 2)

```c
// topic, publisher, writer_listener already created
DDSDataWriter* writer = publisher-&gt;create_datawriter (topic,
    DDS_DATAWRITER_QOS_USE_TOPIC_QOS,
        writer_listener, DDS_STATUS_MASK_ALL);
```

For more information on the general use and manipulation of QosPolicies, see 4.1.7 Getting, Setting, and Comparing QosPolicies on page 172.
7.3.16 Navigating Relationships Among DDS Entities

7.3.16.1 Finding Matching Subscriptions

The following DataWriter operations can be used to get information on the DataReaders that are currently associated with the DataWriter (that is, the DataReaders to which Connext DDS will send the data written by the DataWriter). A subscription consists of information about the DataReader and its associated Subscriber and Topic.

- get_matched_subscriptions()
- get_matched_subscription_data()
- get_matched_subscription_locators()

get_matched_subscriptions() will return a sequence of handles to matched DataReaders. You can use these handles in the get_matched_subscription_data() method to get information about the DataReader such as the values of its QosPolicies, as well as information about its Subscriber and Topic.

get_matched_subscription_locators() retrieves a list of locators for subscriptions currently "associated" with the DataWriter. Matched subscription locators include locators for all those subscriptions in the same DDS domain that have a matching Topic, compatible QoS, and a common partition that the DomainParticipant has not indicated should be "ignored." These are the locators that Connext DDS uses to communicate with matching DataReaders. (See 15.2.1.1 Locator Format on page 770.)

**Note:** In the Modern C++ API, these operations are freestanding functions in the dds::pub or rti::pub namespaces.

You can also get the DATA_WRITER_PROTOCOL_STATUS for matching subscriptions with these operations (see 7.3.6.3 DATA_WRITER_PROTOCOL_STATUS on page 298):

- get_matched_subscription_datawriter_protocol_status()
- get_matched_subscription_datawriter_protocol_status_by_locator()

**Notes:**

- The get_matched_subscriptions() function includes the return of handles of matched DataReaders that are no longer active. All of the handles returned by this function are valid inputs to the get_matched_subscription_data() function.

- Status/data for a matched subscription is kept even if the matched DataReader is not active. Status/data for a matched subscription will be removed only if the DataReader is gone: that is, the DataReader is destroyed and this change is propagated through a discovery update, or the DataReader's DomainParticipant is gone (either gracefully or its liveliness expired and Connext DDS is configured to purge not-alive participants). Once a matched DataReader is gone, its status is
7.3.16 Navigating Relationships Among DDS Entities

deleted. If you try to get the status/data for a matched DataReader that is gone, the 'get status' or 'get data' call will return an error.

- If you want to know which matched DataReaders are not active, use is_matched_subscription_active(). See Table 7.3 DataWriter Operations on page 288.

- DataReaders that have been ignored using the DomainParticipant’s ignore_subscription() operation are not considered to be matched even if the DataReader has the same Topic and compatible QosPolicies. Thus, they will not be included in the list of DataReaders returned by get_matched_subscriptions() or get_matched_subscription_locators(). See 18.4.2 Ignoring Publications and Subscriptions on page 843 for more on ignore_subscription().

- The get_matched_subscription_data() operation does not retrieve the type_code information from built-in-topic data structures. This information is available through the on_data_available() callback (if a DataReaderListener is installed on the SubscriptionBuiltinTopicDataDataReader).

See also: 7.3.16.2 Finding the Matching Subscription’s ParticipantBuiltinTopicData below

7.3.16.2 Finding the Matching Subscription’s ParticipantBuiltinTopicData

get_matched_subscription_participant_data() allows you to get the DDS_ParticipantBuiltinTopicData (see Table 18.1 Participant Built-in Topic’s Data Type (DDS_ParticipantBuiltinTopicData)) of a matched subscription using a subscription handle.

This operation retrieves the information on a discovered DomainParticipant associated with the subscription that is currently matching with the DataWriter. The subscription handle passed into this operation must correspond to a subscription currently associated with the DataWriter. Otherwise, the operation will fail with RETCODE_BAD_PARAMETER. The operation may also fail with RETCODE_PRECONDITION_NOT_MET if the subscription corresponds to the same DomainParticipant to which the DataWriter belongs.

Use get_matched_subscriptions() (see 7.3.16.1 Finding Matching Subscriptions on the previous page) to find the subscriptions that are currently matched with the DataWriter.

7.3.16.3 Finding Related DDS Entities

These operations are useful for obtaining a handle to various related Entities:

- get_publisher()
- get_topic()

get_publisher() returns the Publisher that created the DataWriter. get_topic() returns the Topic with which the DataWriter is associated.
7.3.17 Asserting Liveliness

The `assert_liveliness()` operation can be used to manually assert the liveliness of the `DataWriter` without writing data. This operation is only useful if the kind of 7.5.15 LIVELINESS QosPolicy on page 427 is `MANUAL_BY_PARTICIPANT` or `MANUAL_BY_TOPIC`.

How `DataReaders` determine if `DataWriters` are alive is configured using the 7.5.15 LIVELINESS QosPolicy on page 427. The `lease_duration` parameter of the LIVELINESS QosPolicy is a contract by the `DataWriter` to all of its matched `DataReaders` that it will send a packet within the time value of the `lease_duration` to state that it is still alive.

There are three ways to assert liveliness. One is to have Connext DDS itself send liveliness packets periodically when the kind of LIVELINESS QosPolicy is set to AUTOMATIC. The other two ways to assert liveliness, used when liveliness is set to MANUAL, are to call `write()` to send data or to call the `assert_liveliness()` operation without sending data.

7.3.18 Turbo Mode and Automatic Throttling for DataWriter Performance—Experimental Features

This section describes two experimental features. The `DataWriter` has many QoS settings that can affect the latency and throughput of outgoing data. There are QoS settings to control send window size (see 11.3.2.1 Understanding the Send Queue and Setting its Size on page 697) and settings that allow to aggregate multiple DDS samples together to reduce CPU and bandwidth utilization (see 7.5.2 BATCH QosPolicy (DDS Extension) on page 374 and 7.6 FlowControllers (DDS Extension) on page 470). The choice of settings that provide the best performance depends on several factors, such as the frequency of writing data, the size of the data, or the condition of the network. If these factors do not change over time, you can choose values for those QoS settings that best suit your system. If these factors do change over time in your system, you can use the following properties to let Connext DDS automatically adjust the QoS settings as system conditions change:

- `dds.domain_participant.auto_throttle.enable`: Configures the DomainParticipant to gather internal measurements (during DomainParticipant creation) that are required for the Auto Throttle feature. This allows DataWriters belonging to this DomainParticipant to use the Auto Throttle feature. Default: false.

- `dds.data_writer.auto_throttle.enable`: Enables automatic throttling in the DataWriter so it can automatically adjust the writing rate and the send window size; this minimizes the need for repair DDS samples and improves latency. Default: false. For additional information on automatic throttling, see 7.5.2.4 Turbo Mode: Automatically Adjusting the Number of Bytes in a Batch—Experimental Feature on page 377.

**Note:** This property takes effect only in DataWriters that belong to a DomainParticipant that has set the property `dds.domain_participant.auto_throttle.enable` (described above) to true.
7.4 Publisher/Subscriber QosPolicies

- **dds.data_writer.enable_turbo_mode**: Enables Turbo Mode and adjusts the batch `max_data_bytes` on page 374 (see 7.5.2 BATCH QosPolicy (DDS Extension) on page 374) based on how frequently the `DataWriter` writes data. Default: false. For additional information, see 7.5.2.4 Turbo Mode: Automatically Adjusting the Number of Bytes in a Batch—Experimental Feature on page 377.

The Built-in QoS profile `BuiltinQosLibExp::Generic.AutoTuning` enables both Turbo Mode and Auto Throttling.

### 7.4 Publisher/Subscriber QosPolicies

This section provides detailed information on the QosPolicies associated with a *Publisher*. Note that *Subscribers* have the exact same set of policies. Table 7.2 Publisher QosPolicies provides a quick reference. They are presented here in alphabetical order.

1. 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) below
2. 7.4.2 ENTITYFACTORY QosPolicy on page 348
3. 7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) on page 350
4. 7.4.4 GROUP_DATA QosPolicy on page 353
5. 7.4.5 PARTITION QosPolicy on page 356
6. 7.4.6 PRESENTATION QosPolicy on page 362

#### 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension)

This QosPolicy is used to enable or disable asynchronous publishing, asynchronous batch flushing, and TopicQuery publishing for the *Publisher*.

For each of these features enabled, the *Publisher* will spawn a thread. There is a thread for asynchronous publishing, a thread for asynchronous batch flushing, and a thread for TopicQuery publication.

The asynchronous publisher thread will be shared by all *DataWriters* (belonging to this *Publisher*) that have their 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444 **kind** set to ASYNCHRONOUS. The asynchronous publishing thread will then handle the data transmission chores for those *DataWriters*. This thread will only be spawned when the first of these *DataWriters* is enabled.

The asynchronous publisher thread can be used to reduce the amount of time spent in the user thread to send data. You must use it when sending large data reliably. Large in this context means that the data size is larger than the transport’s `message_size_max`. See also 23.3 Large Data Fragmentation on page 970.

The asynchronous batch flushing thread will be shared by all *DataWriters* (belonging to this *Publisher*) that have batching enabled and `max_flush_delay` different than `DURATION_INFINITE` in 7.5.2 BATCH QosPolicy (DDS Extension) on page 374. This thread will only be spawned when the first of these *DataWriters* is enabled.
The TopicQuery publication thread will be shared by all DataWriters (belonging to this Publisher) that have topic query dispatch enabled in 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455. This thread will only be spawned when the first of these DataWriters is enabled.

This QosPolicy allows you to adjust the asynchronous publishing, the asynchronous batch flushing threads, and the TopicQuery publication threads independently.

Batching and asynchronous publication are independent of one another. Flushing a batch on an asynchronous DataWriter makes it available for sending to the DataWriter's 7.6 FlowControllers (DDS Extension) on page 470. From the point of view of the FlowController, a batch is treated like one large DDS sample.

Connext DDS will sometimes coalesce multiple DDS samples into a single network datagram. For example, DDS samples buffered by a FlowController or sent in response to a negative acknowledgement (NACK) may be coalesced. This behavior is distinct from DDS sample batching. DDS data samples sent by different asynchronous DataWriters belonging to the same Publisher to the same destination will not be coalesced into a single network packet. Instead, two separate network packets will be sent. Only DDS samples written by the same DataWriter and intended for the same destination will be coalesced.

This QosPolicy includes the members in Table 7.19 DDS_AsymmetricalPublisherQosPolicy.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field_Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>disable_asyncronous_write</td>
<td>Disables asynchronous publishing. To write asynchronously, this field must be FALSE (the default).</td>
</tr>
<tr>
<td>DDS_ThreadSettings_t</td>
<td>thread</td>
<td>Settings for the publishing thread. These settings are OS-dependent (see the RTI Connext DDS Core Libraries Platform Notes).</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>disable_asyncronous_batch</td>
<td>Disables asynchronous batch flushing. To flush asynchronously, this field must be FALSE (the default).</td>
</tr>
<tr>
<td>DDS_ThreadSettings_t</td>
<td>asynchronous_batch_thread</td>
<td>Settings for the asynchronous batch flushing thread. These settings are OS-dependent (see the RTI Connext DDS Core Libraries Platform Notes).</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>disable_topic_query_publication</td>
<td>Disables TopicQuery publication. To allow publishing TopicQueries responses, this field must be FALSE (the default).</td>
</tr>
<tr>
<td>DDS_ThreadSettings_t</td>
<td>topic_query_publication_thread</td>
<td>Settings for the TopicQuery publication thread. These settings are OS-dependent (see the RTI Connext DDS Core Libraries Platform Notes).</td>
</tr>
</tbody>
</table>

### 7.4.1.1 Properties

This QosPolicy cannot be modified after the Publisher is created.

Since it is only for Publishers, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.
7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension)

7.4.1.2 Related Property Qos Policies

- `dds.domain_participant.asynchronous_publisher_thread_destruction_timeout`: Maximum time in seconds the DomainParticipant will wait for the destruction of an asynchronous publisher thread. If this timeout expires before the asynchronous publisher thread is destroyed, the DomainParticipant cannot safely release the thread's resources, and it will skip their release. Default: 10 (seconds). Valid values: 1-60 (seconds).

7.4.1.3 Related QosPolicies

- If `disable_asynchronous_write` is TRUE (not the default), then any DataWriters created from this Publisher must have their 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444 kind set to SYNCHRONOUS. (Otherwise `create_datawriter()` will return INCONSISTENT_QOS.)

- If `disable_asynchronous_batch` is TRUE (not the default), then any DataWriters created from this Publisher must have `max_flush_delay` in 7.5.2 BATCH QosPolicy (DDS Extension) on page 374 set to DURATION_INFINITE. (Otherwise `create_datawriter()` will return INCONSISTENT_QOS.)

- DataWriters configured to use the 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432 do not support asynchronous publishing; an error is returned if a multi-channel DataWriter is configured for asynchronous publishing.

- If `disable_topic_query_publication` is TRUE (not the default), then any DataWriters created from this Publisher must have `enable` in 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455) to TRUE. (Otherwise `create_datawriter()` will return INCONSISTENT_QOS.)

7.4.1.4 Applicable DDS Entities

7.2 Publishers on page 271

7.4.1.5 System Resource Considerations

Three threads can potentially be created:

- For asynchronous publishing, system resource usage depends on the activity of the asynchronous thread controlled by the FlowController (see 7.6 FlowControllers (DDS Extension) on page 470).

- For asynchronous batch flushing, system resource usage depends on the activity of the asynchronous thread controlled by `max_flush_delay` in 7.5.2 BATCH QosPolicy (DDS Extension) on page 374.
• For TopicQuery publication, system resource usage depends on the activity of the TopicQuery publication thread controlled by 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455.

7.4.2 ENTITYFACTORY QosPolicy

This QosPolicy controls whether or not child Entities are created in the enabled state.

This QosPolicy applies to the DomainParticipantFactory, DomainParticipants, Publishers, and Subscribers, which act as ‘factories’ for the creation of subordinate Entities. A DomainParticipantFactory is used to create DomainParticipants. A DomainParticipant is used to create both Publishers and Subscribers. A Publisher is used to create DataWriters, similarly a Subscriber is used to create DataReaders.

Entities can be created either in an ‘enabled’ or ‘disabled’ state. An enabled entity can actively participate in communication. A disabled entity cannot be discovered or take part in communication until it is explicitly enabled. For example, Connext DDS will not send data if the write() operation is called on a disabled DataWriter, nor will Connext DDS deliver data to a disabled DataReader. You can only enable a disabled entity. Once an entity is enabled, you cannot disable it, see 4.1.2 Enabling DDS Entities on page 168 about the enable() method.

The ENTITYFACTORY contains only one member, as illustrated in Table 7.20 DDS_EntityFactoryQosPolicy.

Table 7.20 DDS_EntityFactoryQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>autoenable_created_entities</td>
<td>DDS_BOOLEAN_TRUE: enable Entities when they are created</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDS_BOOLEAN_FALSE: do not enable Entities when they are created</td>
</tr>
</tbody>
</table>

The ENTITYFACTORY QosPolicy controls whether the Entities created from the factory are automatically enabled upon creation or are left disabled. For example, if a Publisher is configured to auto-enable created Entities, then all DataWriters created from that Publisher will be automatically enabled.

Note: if an entity is disabled, then all of the child Entities it creates are also created in a disabled state, regardless of the setting of this QosPolicy. However, enabling a disabled entity will enable all of its children if this QosPolicy is set to autoenable child Entities.

Note: an entity can only be enabled; it cannot be disabled after its been enabled.

See 7.4.2.1 Example below for an example of how to set this policy.

There are various reasons why you may want to create Entities in the disabled state:
7.4.2 ENTITYFACTORY QosPolicy

- To get around a “chicken and egg”-type issue. Where you need to have an entity in order to modify it, but you don’t want the entity to be used by Connext DDS until it has been modified.

For example, if you create a DomainParticipant in the enabled state, it will immediately start sending packets to other nodes trying to discover if other Connext DDS applications exist. However, you may want to configure the built-in topic reader listener before discovery occurs. To do this, you need to create a DomainParticipant in the disabled state because once enabled, discovery will occur. If you set up the built-in topic reader listener after the DomainParticipant is enabled, you may miss some discovery traffic.

- You may want to create Entities without having them automatically start to work. This especially pertains to DataReaders. If you create a DataReader in an enabled state and you are using DataReaderListeners, Connext DDS will immediately search for matching DataWriters and call-back the listener as soon as data is published. This may not be what you want to happen if your application is still in the middle of initialization when data arrives.

So typically, you would create all Entities in a disabled state, and then when all parts of the application have been initialized, one would enable all Entities at the same time using the enable() operation on the DomainParticipant, see 4.1.2 Enabling DDS Entities on page 168.

- An entity’s existence is not advertised to other participants in the network until the entity is enabled. Instead of sending an individual declaration packet to other applications announcing the existence of the entity, Connext DDS can be more efficient in bundling multiple declarations into a single packet when you enable all Entities at the same time.

See 4.1.2 Enabling DDS Entities on page 168 for more information about enabled/disabled Entities.

7.4.2.1 Example

The code in Figure 7.25: Configuring a Publisher so that New DataWriters are Disabled on the next page illustrates how to use the ENTITYFACTORY QoS.

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.
7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension)

Figure 7.25: Configuring a Publisher so that New DataWriters are Disabled

```c
DDS_PublisherQos publisher_qos;
// topic, publisher, writer_listener already created
if (publisher->get_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
publisher_qos.entity_factory.autoenable_created_entities
    = DDS_BOOLEAN_FALSE;
if (publisher->set_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Subsequently created DataWriters are created disabled and
// must be explicitly enabled by the user-code
DDSDataWriter* writer = publisher->create_datawriter(topic,
    DDS_DATAWRITER_QOS_DEFAULT, writer_listener, DDS_STATUS_MASK_ALL);
// now do other initialization
// Now explicitly enable the DataWriter, this will allow other
// applications to discover the DataWriter and for this application
// to send data when the DataWriter’s write() method is called
writer->enable();
```

7.4.2.2 Properties

This QosPolicy can be modified at any time.

It can be set differently on the publishing and subscribing sides.

7.4.2.3 Related QosPolicies

This QosPolicy does not interact with any other policies.

7.4.2.4 Applicable DDS Entities

- 9.2 DomainParticipantFactory on page 602
- 9.3 DomainParticipants on page 609
- 7.2 Publishers on page 271
- 8.2 Subscribers on page 486

7.4.2.5 System Resource Considerations

This QosPolicy does not significantly impact the use of system resources.

7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension)

This QoSPolicy is deprecated as of release 6.1.1 and will be removed in a future release.

This QosPolicy controls the creation and use of Exclusive Areas. An exclusive area (EA) is a mutex with built-in deadlock protection when multiple EAs are in use. It is used to provide mutual exclusion among
different threads of execution. Multiple EAs allow greater concurrency among the internal and user threads when executing Connext DDS code.

EAs allow Connext DDS to be multi-threaded while preventing threads from a classical deadlock scenario for multi-threaded applications. EAs prevent a DomainParticipant's internal threads from deadlocking with each other when executing internal code as well as when executing the code of user-registered listener callbacks.

Within an EA, all calls to the code protected by the EA are single threaded. Each DomainParticipant, Publisher and Subscriber represents a separate EA. All DataWriters of the same Publisher and all DataReaders of the same Subscriber share the EA of its parent. This means that the DataWriters of the same Publisher and the DataReaders of the same Subscriber are inherently single threaded.

Within an EA, there are limitations on how code protected by a different EA can be accessed. For example, when data is being processed by user code received in the DataReaderListener of a Subscriber EA, the user code may call the write() function of a DataWriter that is protected by the EA of its Publisher. So you can send data in the function called to process received data. However, you cannot create Entities or call functions that are protected by the EA of the DomainParticipant. See 4.5 Exclusive Areas (EAs) on page 197 for the complete documentation on Exclusive Areas.

With this QoS, you can force a Publisher or Subscriber to share the same EA as its DomainParticipant. Using this capability, the restriction of not being to create Entities in a DataReaderListener's on_data_available() callback is lifted. However, the trade-off is that the application has reduced concurrency through the Entities that share an EA.

Note that the restrictions on calling methods in a different EA only exists for user code that is called in registered Listeners by internal DomainParticipant threads. User code may call all Connext DDS functions for any Entities from their own threads at any time.

The EXCLUSIVE_AREA includes a single member, as listed in Table 7.21 DDS_ExclusiveAreaQosPolicy. For the default value, please see the API Reference HTML documentation.

### Table 7.21 DDS_ExclusiveAreaQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>use_shared_exclusive_area</td>
<td>DDS_BOOLEAN_FALSE: subordinates will not use the same EA</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDS_BOOLEAN_TRUE: subordinates will use the same EA</td>
</tr>
</tbody>
</table>

The implications and restrictions of using a private or shared EA are discussed in 4.5 Exclusive Areas (EAs) on page 197. The basic trade-off is concurrency versus restrictions on which methods can be called in user, listener, callback functions. To summarize:
7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension)

Behavior when the Publisher or Subscriber’s use_shared_exclusive_area is set to FALSE:

- The creation of the Publisher/Subscriber will create an EA that will be used only by the Publisher/Subscriber and the DataWriters/DataReaders that belong to them.
- Consequences: This setting maximizes concurrency at the expense of creating a mutex for the Publisher or Subscriber. In addition, using a separate EA may restrict certain Connext DDS operations (see 4.4.6 Operations Allowed within Listener Callbacks on page 196) from being called from the callbacks of Listeners attached to those Entities and the Entities that they create. This limitation results from a built-in deadlock protection mechanism.

Behavior when the Publisher or Subscriber’s use_shared_exclusive_area is set to TRUE:

- The creation of the Publisher/Subscriber does not create a new EA. Instead, the Publisher/Subscriber, along with the DataWriters/DataReaders that they create, will use a common EA shared with the DomainParticipant.
- Consequences: By sharing the same EA among multiple Entities, you may decrease the amount of concurrency in the application, which can adversely impact performance. However, this setting does use less resources and allows you to call almost any operation on any Entity within a listener callback (see 4.5 Exclusive Areas (EAs) on page 197 for full details).

7.4.3.1 Example

The code in Figure 7.26: Creating a Publisher with a Shared Exclusive Area below illustrates how to change the EXCLUSIVE_AREA policy.

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

Figure 7.26: Creating a Publisher with a Shared Exclusive Area

```c
DDS_PublisherQos publisher_qos;
// domain, publisher_listener have been previously created
if (participant->get_default_publisher_qos(publisher_qos) !=
    DDS_RETCODE_OK) {
    // handle error
}
publisher_qos.exclusive_area.use_shared_exclusive_area = DDS_BOOLEAN_TRUE;
DDSPublisher* publisher = participant->create_publisher(publisher_qos,
    publisher_listener, DDS_STATUS_MASK_ALL);
```

7.4.3.2 Properties

This QosPolicy cannot be modified after the Entity has been created.
It can be set differently on the publishing and subscribing sides.

**7.4.3.3 Related QosPolicies**

This QosPolicy does not interact with any other policies.

**7.4.3.4 Applicable DDS Entities**

- 7.2 Publishers on page 271
- 8.2 Subscribers on page 486

**7.4.3.5 System Resource Considerations**

This QosPolicy affects the use of operating-system mutexes. When `use_shared_exclusive_area` is `FALSE`, the creation of a Publisher or Subscriber will create an operating-system mutex.

**7.4.4 GROUP_DATA QosPolicy**

This QosPolicy provides an area where your application can store additional information related to the Publisher and Subscriber. This information is passed between applications during discovery (see Discovery (Chapter 15 on page 764)) using built-in-topics (see Built-In Topics (Chapter 18 on page 831)). How this information is used will be up to user code. Connext DDS does not do anything with the information stored as GROUP_DATA except to pass it to other applications.

Use cases are often application-to-application identification, authentication, authorization, and encryption purposes. For example, applications can use this QosPolicy to send security certificates to each other for RSA-type security.

The value of the GROUP_DATA QosPolicy is sent to remote applications when they are first discovered, as well as when the Publisher or Subscriber’s `set_gos()` method is called after changing the value of the GROUP_DATA. User code can set listeners on the built-in DataReaders of the built-in Topics used by Connext DDS to propagate discovery information. Methods in the built-in topic listeners will be called whenever new DomainParticipants, DataReaders, and DataWriters are found. Within the user callback, you will have access to the GROUP_DATA that was set for the associated Publisher or Subscriber.

Currently, GROUP_DATA of the associated Publisher or Subscriber is only propagated with the information that declares a DataWriter or DataReader. Thus, you will need to access the value of GROUP_DATA through DDS_PublicationBuiltinTopicData or DDS_SubscriptionBuiltinTopicData (see Built-In Topics (Chapter 18 on page 831)).

The structure for the GROUP_DATA QosPolicy includes just one field, as seen in Table 7.22 DDS_GroupDataQosPolicy. The field is a sequence of octets that translates to a contiguous buffer of bytes whose contents and length is set by the user. The maximum size for the data are set in the 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654.
### 7.4.4 GROUP_DATA QosPolicy

#### Table 7.22 DDS_GroupDataQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_OctetSeq</td>
<td>value</td>
<td>Empty by default</td>
</tr>
</tbody>
</table>

This policy is similar to the 7.5.30 USER_DATA QosPolicy on page 465 and 5.2.1 TOPIC_DATA QosPolicy on page 224 that apply to other types of Entities.

#### 7.4.4.1 Example

One possible use of GROUP_DATA is to pass some credential or certificate that your subscriber application can use to accept or reject communication with the DataWriters that belong to the Publisher (or vice versa, where the publisher application can validate the permission of DataReaders of a Subscriber to receive its data). The value of the GROUP_DATA of the Publisher is propagated in the ‘group_data’ field of the DDS_PublicationBuiltinTopicData that is sent with the declaration of each DataWriter. Similarly, the value of the GROUP_DATA of the Subscriber is propagated in the ‘group_data’ field of the DDS_SubscriptionBuiltinTopicData that is sent with the declaration of each DataReader.

When Connext DDS discovers a DataWriter/DataReader, the application can be notified of the discovery of the new entity and retrieve information about the DataWriter/DataReader QoS by reading the DCPSPublication or DCPSSubscription built-in topics (see Built-In Topics (Chapter 18 on page 831)). Your application can then examine the GROUP_DATA field in the built-in Topic and decide whether or not the DataWriter/DataReader should be allowed to communicate with local DataReaders/DataWriters. If communication is not allowed, the application can use the DomainParticipant's `ignore_publication()` or `ignore_subscription()` operation to reject the newly discovered remote entity as one with which the application allows Connext DDS to communicate. See Figure 18.2: Ignoring Publications on page 844 for an example of how to do this.

The code in Figure 7.27: Creating a Publisher with GROUP_DATA below illustrates how to change the GROUP_DATA policy.

**Note:**

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

#### Figure 7.27: Creating a Publisher with GROUP_DATA

```c
DDS_PublisherQos publisher_qos;
int i = 0;
// Bytes that will be used for the group data. In this case, 8 bytes
// of some information that is meaningful to the user application
char myGroupData[GROUP_DATA_SIZE] =
    { 0x34, 0xaa, 0xfe, 0x31, 0x7a, 0xf2, 0x34, 0xaa};
// assume domainparticipant and publisher_listener already created
```
if (participant->get_default_publisher_qos(publisher_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Must set the size of the sequence first
publisher_qos.group_data.value.maximum(GROUP_DATA_SIZE);
publisher_qos.group_data.value.length(GROUP_DATA_SIZE);
for (i = 0; i < GROUP_DATA_SIZE; i++) {
    publisher_qos.group_data.value[i] = myGroupData[i]
}
DDSPublisher* publisher = participant->create_publisher( publisher_qos,
    publisher_listener, DDS_STATUS_MASK_ALL);

7.4.4.2 Properties

This QosPolicy can be modified at any time.

It can be set differently on the publishing and subscribing sides.

7.4.4.3 Related QosPolicies

- 5.2.1 TOPIC_DATA QosPolicy on page 224
- 7.5.30 USER_DATA QosPolicy on page 465
- 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654

7.4.4.4 Applicable DDS Entities

- 7.2 Publishers on page 271
- 8.2 Subscribers on page 486

7.4.4.5 System Resource Considerations

The maximum size of the GROUP_DATA is set in the publisher_group_data_max_length and subscriber_group_data_max_length fields of the 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654. Because Connext DDS will allocate memory based on this value, you should only increase this value if you need to. If your system does not use GROUP_DATA, then you can set this value to zero to save memory. Setting the value of the GROUP_DATA QosPolicy to hold data longer than the value set in the [publisher/subscriber]_group_data_max_length fields will result in failure and an INCONSISTENT_QOS_POLICY return code.

However, should you decide to change the maximum size of GROUP_DATA, you must make certain that all applications in the DDS domain have changed the value of [publisher/subscriber]_group_data_max_length to be the same. If two applications have different limits on the size of GROUP DATA, and one application sets the GROUP_DATA QosPolicy to hold data that is greater than the maximum size set by another application, then the matching DataWriters and DataReaders of the Publisher and Subscriber between the two applications will not connect. This is also true for the TOPIC_DATA (5.2.1 TOPIC_
The PARTITION QoS provides another way to control which DataWriters will match—and thus communicate with—which DataReaders. It can be used to prevent DataWriters and DataReaders that would have otherwise matched with the same Topic and compatible QoS Policies from talking to each other. Much in the same way that only applications within the same DDS domain will communicate with each other, only DataWriters and DataReaders that belong to the same partition can talk to each other.

The PARTITION QoS applies to Publishers and Subscribers, therefore the DataWriters and DataReaders belong to the partitions as set on the Publishers and Subscribers that created them. The mechanism implementing the PARTITION QoS is relatively lightweight, and membership in a partition can be dynamically changed. Unlike the creation and destruction of DomainParticipants, there is no spawning and killing of threads or allocation and deallocation of memory when Publishers and Subscribers add or remove themselves from partitions.

The PARTITION QoS consists of a set of partition names that identify the partitions of which the Entity is a member. These names are simply strings, and DataWriters and DataReaders are considered to be in the same partition if they have at least one partition name in common in the PARTITION QoS set on their Publishers or Subscribers. By default, Publishers and Subscribers belong to a single partition whose name is the empty string, "".

Conceptually each partition name can be thought of as defining a “visibility plane” within the DDS domain. DataWriters will make their data available on all the visibility planes that correspond to its Publisher’s partition names, and the DataReaders will see the data that is placed on any of the visibility planes that correspond to its Subscriber’s partition names.

Figure 7.28: Controlling Visibility of Data with the PARTITION QoS on the next page illustrates the concept of PARTITION QoS. In this figure, all DataWriters and DataReaders belong to the same DDS domain and refer to the same Topic. DataWriter1 is configured to belong to three partitions: partition_A, partition_B, and partition_C. DataWriter2 belongs to partition_C and partition_D.
Figure 7.28: Controlling Visibility of Data with the PARTITION QoS

Similarly, DataReader1 is configured to belong to partition_A and partition_B, and DataReader2 belongs only to partition_C. Given this topology, the data written by DataWriter1 is visible in partitions A, B, and C. The oval tagged with the number “1” represents one DDS data sample written by DataWriter1.

Similarly, the data written by DataWriter2 is visible in partitions C and D. The oval tagged with the number “2” represents one DDS data sample written by DataWriter2.

The result is that the data written by DataWriter1 will be received by both DataReader1 and DataReader2, but the data written by DataWriter2 will only be visible by DataReader2.

Publishers and Subscribers always belong to a partition. By default, Publishers and Subscribers belong to a single partition whose name is the empty string, “”. If you set the PARTITION QoS to be an empty set, Connext DDS will assign the Publisher or Subscriber to the default partition, “”. Thus, for the example above, without using the PARTITION QoS, DataReaders 1 and 2 would have received all DDS data DDS samples written by DataWriters 1 and 2.

7.4.5.1 Rules for PARTITION Matching

On the Publisher side, the PARTITION QosPolicy associates a set of strings (partition names) with the Publisher. On the Subscriber side, the application also uses the PARTITION QoS to associate partition names with the Subscriber.

Taking into account the PARTITION QoS, a DataWriter will communicate with a DataReader if and only if the following conditions apply:

1. The DataWriter and DataReader belong to the same DDS domain. That is, their respective DomainParticipants are bound to the same DDS domain ID (see 9.3.1 Creating a DomainParticipant on page 615).
2. The DataWriter and DataReader have matching Topics. That is, each is associated with a Topic with the same topic_name and data type.

3. The QoS offered by the DataWriter is compatible with the QoS requested by the DataReader.

4. The application has not used the ignore_participant(), ignore_datareader(), or ignore_datawriter() APIs to prevent the association (see 18.4 Restricting Communication—Ignoring Entities on page 841).

5. The Publisher to which the DataWriter belongs and the Subscriber to which the DataReader belongs must have at least one matching partition name.

The last condition reflects the visibility of the data introduced by the PARTITION QoS. Matching partition names is done by string comparison, thus partition names are case sensitive.

Note: Failure to match partitions is not considered an incompatible QoS and does not trigger any listeners or change any status conditions.

7.4.5.2 Pattern Matching for PARTITION Names

You may also add strings that are regular expressions (as defined in the POSIX fnmatch API (1003.2-1992 Section B.6)) to the PARTITION QosPolicy. A regular expression does not define a set of partitions to which the Publisher or Subscriber belongs, as much as it is used in the partition matching process to see if a remote entity has a partition name that would be matched with the regular expression. That is, the regular expressions in the PARTITION QoS of a Publisher are never matched against those found in the PARTITION QoS of a Subscriber. Regular expressions are always matched against “concrete” partition names. Thus, a concrete partition name may not contain any reserved characters that are used to define regular expressions, for example ‘*’, ‘.’, ‘+’, etc.

For more on regular expressions, see 5.4.6.5 SQL Extension: Regular Expression Matching on page 241.

If a PARTITION QoS only contains regular expressions, then the Publisher or Subscriber will be assigned automatically to the default partition with the empty string name (““). Thus, do not be fooled into thinking that a PARTITION QoS that only contains the string “*” matches another PARTITION QoS that only contains the string “*”. Yes, the Publisher will match the Subscriber, but it is because they both belong to the default “*” partition.

DataWriters and DataReaders are considered to have a partition in common if the sets of partitions that their associated Publishers and Subscribers have defined have:

At least one concrete partition name in common

A regular expression in one Entity that matches a concrete partition name in another Entity

The programmatic representation of the PARTITION QoS is shown in Table 7.23 DDS_PartitionQosPolicy. The QosPolicy contains the single string sequence, name. Each element in the sequence can be a concrete name or a regular expression. The Entity will be assigned to the default ““ partition if the sequence is empty.
7.4.5 PARTITION QosPolicy

Table 7.23 DDS_PartitionQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_StringSeq</td>
<td>name</td>
<td>Empty by default.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>There can be up to 64 names, with a maximum of 256 characters summed across all names.</td>
</tr>
</tbody>
</table>

You can have one long partition string of 256 chars, or multiple shorter strings that add up to 256 or less characters. For example, you can have one string of 4 chars and one string of 252 chars.

7.4.5.3 Example

Since the set of partitions for a Publisher or Subscriber can be dynamically changed, the Partition QosPolicy is useful to control which DataWriters can send data to which DataReaders and vice versa—even if all of the DataWriters and DataReaders are for the same topic. This facility is useful for creating temporary separation groups among Entities that would otherwise be connected to and exchange data each other.

Note when using Partitions and Durability: If a Publisher changes partitions after startup, it is possible for a reliable, late-joining DataReader to receive data that was written for both the original and the new partition. For example, suppose a DataWriter with TRANSIENT_LOCAL Durability initially writes DDS samples with Partition A, but later changes to Partition B. In this case, a reliable, late-joining DataReader configured for Partition B will receive whatever DDS samples have been saved for the DataWriter. These may include DDS samples which were written when the DataWriter was using Partition A.

The code in Figure 7.29: Setting Partition Names on a Publisher on the next page illustrates how to change the PARTITION policy.

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.
The ability to dynamically control which *DataWriters* are matched to which *DataReaders* (of the same *Topic*) offered by the PARTITION QoS can be used in many different ways. Using partitions, connectivity can be controlled based on location-based partitioning, access-control groups, purpose, or a combination of these and other application-defined criteria. We will examine some of these options via concrete examples.

**Example of location-based partitions.** Assume you have a set of *Topics* in a traffic management system such as “TrafficAlert,” “AccidentReport,” and “CongestionStatus.” You may want to control the visibility of these *Topics* based on the actual location to which the information applies. You can do this by placing the *Publisher* in a partition that represents the area to which the information applies. This can be done using a string that includes the city, state, and country, such as “USA/California/Santa Clara.” A *Subscriber* can then choose whether it wants to see the alerts in a single city, the accidents in a set of states, or the congestion status across the US. Some concrete examples are shown in Table 7.24 Example of Using Location-Based Partitions.

<table>
<thead>
<tr>
<th>Publisher Partitions</th>
<th>Subscriber Partitions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify a single partition name using the pattern: <code>&lt;country&gt;/&lt;state&gt;/&lt;city&gt;</code></td>
<td>Specify multiple partition names, one per region of interest</td>
<td>Limits the visibility of the data to Subscribers that express interest in the geographical region.</td>
</tr>
<tr>
<td>“USA/California/Santa Clara”</td>
<td>(Subscriber participant is irrelevant here.)</td>
<td>Send only information for Santa Clara, California.</td>
</tr>
</tbody>
</table>
Table 7.24 Example of Using Location-Based Partitions

<table>
<thead>
<tr>
<th>Publisher Partitions</th>
<th>Subscriber Partitions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;USA/California/Santa Clara&quot;</td>
<td>Receive only information for Santa Clara, California.</td>
<td></td>
</tr>
<tr>
<td>&quot;USA/California/Santa Clara&quot; &quot;USA/California/Sunnyvale&quot;</td>
<td>Receive information for Santa Clara or Sunnyvale, California.</td>
<td></td>
</tr>
<tr>
<td>&quot;USA/California/<em>&quot; &quot;USA/Nevada/</em>&quot;</td>
<td>Receive information for California or Nevada.</td>
<td></td>
</tr>
<tr>
<td>&quot;USA/California/<em>&quot; &quot;USA/Nevada/</em>&quot; &quot;USA/California/Santa Clara&quot; &quot;USA/Nevada/Las Vegas&quot;</td>
<td>Receive information for California and two cities in Nevada.</td>
<td></td>
</tr>
</tbody>
</table>

(Publisher partition is irrelevant here.)

Example of access-control group partitions. Suppose you have an application where access to the information must be restricted based on reader membership to access-control groups. You can map this group-controlled visibility to partitions by naming all the groups (e.g. executives, payroll, financial, general-staff, consultants, external-people) and assigning the Publisher to the set of partitions that represents which groups should have access to the information. The Subscribers specify the groups to which they belong, and the partition-matching behavior will ensure that the information is only distributed to Subscribers belonging to the appropriate groups. Some concrete examples are shown in Table 7.25 Example of Access-Control Group Partitions.

Table 7.25 Example of Access-Control Group Partitions

<table>
<thead>
<tr>
<th>Publisher Partitions</th>
<th>Subscriber Partitions</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specify several partition names, one per group that is allowed access: &quot;payroll&quot; &quot;financial&quot; (Publisher participant is irrelevant here.)</td>
<td>Specify multiple partition names, one per group to which the Subscriber belongs. &quot;executives&quot; &quot;financial&quot; (Subscriber participant is irrelevant here.)</td>
<td>Limits the visibility of the data to Subscribers that belong to the access-groups specified by the Publisher. Makes information available only to Subscribers that have access to either financial or payroll information. Gain access to information that is intended for executives or people with access to the finances.</td>
</tr>
</tbody>
</table>

A slight variation of this pattern could be used to confine the information based on security levels.

Example of purpose-based partitions: Assume an application containing subsystems that can be used for multiple purposes, such as training, simulation, and real use. In some occasions it is convenient to be able to dynamically switch the subsystem from operating in the “simulation world” to the “training world” or to the “real world.” For supervision purposes, it may be convenient to observe multiple worlds, so that you can compare the each one’s results. This can be accomplished by setting a partition name in the Publisher that represents the “world” to which it belongs and a set of partition names in the Subscriber that model the worlds that it can observe.
7.4.5.4 Properties

This QosPolicy can be modified at any time.

Strictly speaking, this QosPolicy does not have request-offered semantics, although it is matched between DataWriters and DataReaders, and communication is established only if there is a match between partition names.

7.4.5.5 Related QosPolicies

- 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654.

7.4.5.6 Applicable DDS Entities

- 7.2 Publishers on page 271
- 8.2 Subscribers on page 486

7.4.5.7 System Resource Considerations

Partition names are propagated along with the declarations of the DataReaders and the DataWriters and can be examined by user code through built-in topics (see Built-In Topics (Chapter 18 on page 831)). Thus the sum-total length of the partition names will impact the bandwidth needed to transmit those declarations, as well as the memory used to store them.

The maximum number of partitions and the maximum number of characters that can be used for the sum-total length of all partition names are configured using the max_partitions and max_partition_cumulative_characters fields of the 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654. Setting more partitions or using longer names than allowed by those limits will result in failure and an INCONSISTENT_QOS_POLICY return code.

However, should you decide to change the maximum number of partitions or maximum cumulative length of partition names, then you must make certain that all applications in the DDS domain have changed the values of max_partitions and max_partition_cumulative_characters to be the same. If two applications have different values for those settings, and one application sets the PARTITION QosPolicy to hold more partitions or longer names than set by another application, then the matching DataWriters and DataReaders of the Publisher and Subscriber between the two applications will not connect. This similar to the restrictions for the GROUP_DATA (7.4.4 GROUP_DATA QosPolicy on page 353), USER_DATA (7.5.30 USER_DATA QosPolicy on page 465), and TOPIC_DATA (5.2.1 TOPIC_DATA QosPolicy on page 224) QosPolicies.

7.4.6 PRESENTATION QosPolicy

Usually DataReaders will receive data in the order that it was sent by a DataWriter. In addition, data is presented to the DataReader as soon as the application receives the next value expected.
Sometimes, you may want a set of data for the same DataWriter or different Data Writers to be presented to the receiving Data Reader(s) only after ALL the elements of the set have been received, but not before. You may also want the data to be presented in a different order than it was received. Specifically, for keyed data, you may want Connext DDS to present the data in keyed or instance order.

The Presentation Qos Policy allows you to specify different scopes of presentation: within a Data Writer, across instances of a Data Writer, and even across different Data Writers of a Publisher. It also controls whether or not a set of changes within the scope must be delivered at the same time or delivered as soon as each element is received. The structure used is shown in Table 7.26 DDS_PresentationQosPolicy.

Table 7.26 DDS_PresentationQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Presentation_QosPolicyAccessScopeKind</td>
<td>access_scope</td>
<td>Determines the largest scope spanning the entities for which the ordered_access and coherent_access of samples can be preserved (if coherent_access and/or ordered_access are TRUE). If both coherent_access and ordered_access are FALSE, access_scope’s setting has no effect.</td>
</tr>
<tr>
<td>DDS_Presentation_QosPolicyAccessScopeKind</td>
<td>coerent_access</td>
<td>Controls whether Connext DDS will preserve the groupings of changes made by the publishing application by means of begin_coherent_changes() and end_coherent_changes().</td>
</tr>
<tr>
<td>DDS_Presentation_QosPolicyAccessScopeKind</td>
<td>ordered_access</td>
<td>Controls whether Connext DDS will preserve the order of changes.</td>
</tr>
</tbody>
</table>

- DDS_INSTANCEPRESENTATION_QOS (default): Scope spans only a single instance. Changes to one instance need not be coherent nor ordered with respect to changes to any other instance. Order and coherent changes apply to each instance separately.
- DDS_TOPICPRESENTATION_QOS: Scope spans all instances within the same DataWriter, but not across instances in different Data Writers.
- DDS_GROUPPRESENTATION_QOS: Scope spans all instances belonging to Data Writers within the same Publisher.
- DDS_HIGHEST_OFFEREDPRESENTATION_QOS: Only applies to Subscribers. With this setting, the Subscriber will use the access scope specified by each remote Publisher.

- DDS_BOOLEAN_FALSE (default): Coherency is not preserved. The value of access_scope is ignored.
- DDS_BOOLEAN_TRUE: Changes made to instances within each Data Writer will be available to the Data Reader as a coherent set, based on the value of access_scope.
### 7.4.6.1 Coherent Access

A 'coherent set' is a set of DDS data-sample modifications that must be propagated in such a way that they are interpreted at the receiver's side as a consistent set; that is, the receiver will only be able to access the data after all the modifications in the set are available at the subscribing end.

Coherency enables a publishing application to change the value of several data-instances and have those changes be seen atomically (as a cohesive set) by the readers.

Setting `coherent_access` to TRUE only behaves as described in the DDS specification when the `DataWriter` and `DataReader` are configured for reliable delivery. Non-reliable `DataReaders` will never receive DDS samples that belong to a coherent set.

To send a coherent set of DDS data samples, the publishing application uses the `Publisher's begin_coherent_changes()` and `end_coherent_changes()` operations (see 7.3.10 Writing Coherent Sets of DDS Data Samples on page 316).

If `coherent_access` is TRUE, then the `access_scope` controls the maximum extent of the coherent changes, as follows:

- If `access_scope` is INSTANCE, the behavior is the same as TOPIC.
- If `access_scope` is TOPIC, then coherent changes done by a `DataWriter` (indicated by their enclosure within calls to `begin_coherent_changes()` and `end_coherent_changes()`) will be made available as a unit to each remote `DataReader` independently. That is, changes made to instances within each individual `DataWriter` will be presented as a unit. They will not be grouped with changes made to instances belonging to a different `DataWriter`. 

---

#### Table 7.26 DDS_PresentationQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>drop_incomplete_coherent_set</td>
<td>Indicates whether a <code>DataReader</code> should drop (and report as lost) samples from an incomplete coherent set (one for which not all the samples were received):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DDS_BOOLEAN_FALSE: The <code>DataReader</code> will not drop samples that are part of an incomplete coherent set.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DDS_BOOLEAN_TRUE (default): The <code>DataReader</code> will drop samples that are part of an incomplete coherent set. Such samples are reported as lost, with the reason LOST_BY_INCOMPLETE_COHERENT_SET, in the 8.3.7.7 SAMPLE_LOST Status on page 527.</td>
</tr>
</tbody>
</table>

Note that a coherent set is also considered incomplete if some of its samples are filtered by content or time on the `DataWriter` side.

Samples from an incomplete coherent set have `incomplete_coherent_set` in the `coherent_set_info` field in the 8.4.6 The SampleInfo Structure on page 565 set to TRUE.
If access\_scope is GROUP, coherent changes made to instances through a set of DataWriters attached to a common Publisher are presented as a unit to the DataReaders within a Subscriber.

### 7.4.6.2 Ordered Access

If ordered\_access is TRUE, then access\_scope controls the scope of the order in which DDS samples are presented to the subscribing application, as follows:

- If access\_scope is INSTANCE, the relative order of DDS samples sent by a DataWriter is only preserved on a per-instance basis. If two DDS samples refer to the same instance (identified by Topic and a particular value for the key) then the order in which they are stored in the DataReader’s queue is consistent with the order in which the changes occurred. However, if the two DDS samples belong to different instances, the order in which they are presented may or may not match the order in which the changes occurred.

- If access\_scope is TOPIC, the relative order of DDS samples sent by a DataWriter is preserved for all DDS samples of all instances. The coherent grouping and/or order in which DDS samples appear in the DataReader’s queue is consistent with the grouping/order in which the changes occurred in that DataWriter—even if the DDS samples affect different instances.

- If access\_scope is GROUP, the scope spans all instances belonging to DataWriters within the same Publisher—even if they are instances of different topics. Changes made to instances via DataWriters attached to the same Publisher are presented to the DataReaders within a Subscriber in the same order in which they occurred.

- If access\_scope is HIGHEST\_OFFERED, the Subscriber will use the access scope specified by each remote Publisher.

If the Subscriber's access\_scope is GROUP or HIGHEST\_OFFERED and ordered\_access is TRUE, the application is required to use the Subscriber's begin\_access() and end\_access() operations to access the DDS samples in order across DataWriters of the same group (a Publisher with access\_scope of GROUP). If you do not use these operations, the data may not be ordered across DataWriters. See 8.2.5 Beginning and Ending Group-Ordered Access on page 499 for additional details.

Ultimately, the data stored in the DataReader queue is accessed by the DataReader’s read()/take() APIs. The application does not have to access the DDS data samples in the order indicated by the combination of access\_scope and ordered\_access. How the application actually gets the data from the DataReader is ultimately under the control of the user code. See 8.4 Using DataReaders to Access Data (Read & Take) on page 553 for additional details.

### 7.4.6.3 Example

Coherency is useful in cases where the grouping matters across multiple Topics of a single Publisher. For example, consider an "Aircraft State" Publisher with two DataWriters, one for a Topic representing the altitude and the other for a Topic representing the velocity vector. If both altitude and velocity are changed
for a given aircraft in the producer application, it may be significant to communicate those values in a way the reader can see both together as a group; otherwise, a consumer application may, for example, erroneously interpret that an aircraft is on a collision course.

Ordered access is useful when you need to ensure that DDS samples appear on the DataReader’s queue in the order sent by one or multiple DataWriters within the same Publisher.

To illustrate the effect of the PRESENTATION QosPolicy with TOPIC and INSTANCE access scope, assume the following sequence of DDS samples was written by the DataWriter: \{A1, B1, C1, A2, B2, C2\}. In this example, A, B, and C represent different instances (i.e., different keys). Assume all of these DDS samples have been propagated to the DataReader’s history queue before your application invokes the read() operation. The DDS data-sample sequence returned depends on how the PRESENTATION QoS is set, as shown in Table 7.27 Effect of ordered_access for access_scope INSTANCE and TOPIC.

### Table 7.27 Effect of ordered_access for access_scope INSTANCE and TOPIC

<table>
<thead>
<tr>
<th>PRESENTATION QoS</th>
<th>Sequence retrieved via “read()”. Order sent was {A1, B1, C1, A2, B2, C2}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered_access = FALSE access_scope = &lt;any&gt;</td>
<td>Any order is possible. For example, {A1,A2,B1,B2,C1,C2}, {A1, B1, C1, A2, B2, C2}, and {C1,B2,A1,A2,B1,C2}</td>
</tr>
<tr>
<td>ordered_access = TRUE access_scope = INSTANCE</td>
<td>Order is preserved per instance. Multiple orders are possible. For example, {A1,A2,B1,B2,C1,C2} or {A1, B1, C1, A2, B2, C2} or {B1,B2,A1,A2,C1,C2}. Recall that coherent_access by INSTANCE does not apply, but ordered_access by INSTANCE does. So for any given instance, the samples are ordered (B1 must come before B2, for example), but Connext DDS does not need to deliver all changes to the instance atomically.</td>
</tr>
<tr>
<td>ordered_access = TRUE access_scope = TOPIC</td>
<td>{A1, B1, C1, A2, B2, C2}</td>
</tr>
</tbody>
</table>

To illustrate the effect of a PRESENTATION QosPolicy with GROUP access_scope, assume the following sequence of DDS samples was written by two DataWriters, W1 and W2, within the same Publisher: \{(W1,A1), (W2,B1), (W1,C1), (W2,A2), (W1,B2), (W2,C2)\}. As in the previous example, A, B, and C represent different instances (i.e., different keys). With access_scope set to INSTANCE or TOPIC, the middleware cannot guarantee that the application will receive the DDS samples in the same order they were published by W1 and W2. With access_scope set to GROUP, the middleware is able to provide the DDS samples in order to the application as long as the read()/take() operations are invoked within a
begin_access()/end_access() block (see 8.2.5 Beginning and Ending Group-Ordered Access on page 499).

### Table 7.28 Effect of ordered_access for access_scope GROUP

<table>
<thead>
<tr>
<th>PRESENTATION QoS</th>
<th>Sequence retrieved via “read()”.</th>
<th>Order sent was {(W1,A1), (W2,B1), (W1,C1), (W2,A2), (W1,B2), (W2,C2)}</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered_access = FALSE or access_scope = TOPIC or INSTANCE</td>
<td>The order across DataWriters will not be preserved. DDS samples may be delivered in multiple orders. For example:</td>
<td>{(W1,A1), (W1,C1), (W2,B1), (W2,A2), (W2,C2)} &lt;br&gt; {(W1,A1), (W2,B1), (W1,B2), (W1,C1), (W2,A2), (W2,C2)}</td>
</tr>
<tr>
<td>ordered_access = TRUE access_scope = GROUP</td>
<td>DDS samples are delivered in the same order they were published:</td>
<td>{(W1,A1), (W2,B1), (W1,C1), (W2,A2), (W1,B2), (W2,C2)}</td>
</tr>
</tbody>
</table>

### 7.4.6.4 Properties

This QosPolicy cannot be modified after the Publisher or Subscriber is enabled.

This QoS must be set compatibly between the DataWriter’s Publisher and the DataReader’s Subscriber. The compatible combinations are shown in Table 7.29 Valid Combinations of ordered_access and access_scope, with Subscriber’s ordered_access = False and Table 7.30 Valid Combinations of ordered_access and access_scope, with Subscriber’s ordered_access = True for ordered_access and Table 7.31 Valid Combinations of Presentation Coherent Access and Access Scope for coherent_access.

### Table 7.29 Valid Combinations of ordered_access and access_scope, with Subscriber’s ordered_access = False

<table>
<thead>
<tr>
<th>{ordered_access/access_scope}</th>
<th>False/Instance</th>
<th>False/Topic</th>
<th>False/Group</th>
<th>False/Highest</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publisher offers:</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>False/Instance</td>
<td>compatible</td>
<td>incompatible</td>
<td>incompatible</td>
<td>compatible</td>
</tr>
<tr>
<td>False/Topic</td>
<td>compatible</td>
<td>compatible</td>
<td>incompatible</td>
<td>compatible</td>
</tr>
<tr>
<td>False/Group</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
</tr>
<tr>
<td>True/Instance</td>
<td>compatible</td>
<td>incompatible</td>
<td>incompatible</td>
<td>compatible</td>
</tr>
<tr>
<td>True/Topic</td>
<td>compatible</td>
<td>compatible</td>
<td>incompatible</td>
<td>compatible</td>
</tr>
<tr>
<td>True/Group</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
</tr>
</tbody>
</table>
### Table 7.30 Valid Combinations of ordered_access and access_scope, with Subscriber's ordered_access = True

<table>
<thead>
<tr>
<th>ordered_access/access_scope</th>
<th>Subscriber Requests:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>True/Instance</td>
</tr>
<tr>
<td>Publisher offers:</td>
<td></td>
</tr>
<tr>
<td>False/Instance</td>
<td>incompatible</td>
</tr>
<tr>
<td>False/Topic</td>
<td>incompatible</td>
</tr>
<tr>
<td>False/Group</td>
<td>incompatible</td>
</tr>
<tr>
<td>True/Instance</td>
<td>compatible</td>
</tr>
<tr>
<td>True/Topic</td>
<td>compatible</td>
</tr>
<tr>
<td>True/Group</td>
<td>compatible</td>
</tr>
</tbody>
</table>

### Table 7.31 Valid Combinations of Presentation Coherent Access and Access Scope

<table>
<thead>
<tr>
<th>coherent_access/access_scope</th>
<th>Subscriber requests:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>False/Instance</td>
</tr>
<tr>
<td>Publisher offers:</td>
<td></td>
</tr>
<tr>
<td>False/Instance</td>
<td>compatible</td>
</tr>
<tr>
<td>False/Topic</td>
<td>compatible</td>
</tr>
<tr>
<td>True/Instance</td>
<td>compatible</td>
</tr>
<tr>
<td>True/Topic</td>
<td>compatible</td>
</tr>
</tbody>
</table>

### 7.4.6.5 Related QosPolicies

- The 7.5.8 DESTINATION_ORDER QosPolicy on page 408 is closely related and also affects the ordering of DDS data samples on a per-instance basis when there are multiple DataWriters.

- The 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570 may be used to configure the DDS sample ordering process in the Subscribers configured with GROUP or HIGHEST_OFFERED access_scope.

### 7.4.6.6 Applicable DDS Entities

- 7.2 Publishers on page 271
- 8.2 Subscribers on page 486
7.4.6.7 System Resource Considerations

The use of this policy does not significantly impact the usage of resources.

7.5 DataWriter QosPolicies

This section provides detailed information about the QosPolicies associated with a DataWriter. Table 7.18 DataWriter QosPolicies provides a quick reference. They are presented here in alphabetical order.

- 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on the next page
- 7.5.2 BATCH QosPolicy (DDS Extension) on page 374
- 7.5.3 DATA_REPRESENTATION QosPolicy on page 380
- 7.5.4 DATATAG QosPolicy on page 388
- 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389
- 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401
- 7.5.7 DEADLINE QosPolicy on page 405
- 7.5.8 DESTINATION_ORDER QosPolicy on page 408
- 7.5.9 DURABILITY QosPolicy on page 411
- 7.5.10 DURABILITY SERVICE QosPolicy on page 416
- 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418
- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.13 LATENCYBUDGET QoS Policy on page 425
- 7.5.14 LIFESPAN QoS Policy on page 425
- 7.5.15 LIVELINESS QosPolicy on page 427
- 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432
- 7.5.17 OWNERSHIP QosPolicy on page 434
- 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438
- 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439
- 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444
- 7.5.21 RELIABILITY QosPolicy on page 447
- 7.5.22 RESOURCE_LIMITS QosPolicy on page 451
- 7.5.23 SERVICE QosPolicy (DDS Extension) on page 454
- 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455
- 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458
7.5.1 AVAILABILITY QosPolicy (DDS Extension)

This QoS policy configures the availability of data and it is used in the context of two features:

- Collaborative DataWriters (7.5.1.1 Availability QoS Policy and Collaborative DataWriters on the next page)
- Required Subscriptions (7.5.1.2 Availability QoS Policy and Required Subscriptions on page 372)

It contains the members listed in Table 7.32 DDS_AvailabilityQosPolicy.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>enable_required_subscriptions</td>
<td>Enables support for required subscriptions in a DataWriter. If set to TRUE, history kind must be KEEP_ALL and durability writer_depth must be left set to AUTO. For Collaborative DataWriters: Not applicable. For Required Subscriptions: See Table 7.35 Configuring Required Subscriptions with DDS_AvailabilityQosPolicy.</td>
</tr>
<tr>
<td>struct DDS_Duration_t</td>
<td>max_data_availability_waiting_time</td>
<td>Defines how much time to wait before delivering a DDS sample to the application without having received some of the previous DDS samples. For Collaborative DataWriters: See Table 7.34 Configuring Collaborative DataWriters with DDS_AvailabilityQosPolicy. For Required Subscriptions: Not applicable.</td>
</tr>
<tr>
<td>struct DDS_Duration_t</td>
<td>max_endpoint_availability_waiting_time</td>
<td>Defines how much time to wait to discover DataWriters providing DDS samples for the same data source. For Collaborative DataWriters: See Table 7.34 Configuring Collaborative DataWriters with DDS_AvailabilityQosPolicy. For Required Subscriptions: Not applicable.</td>
</tr>
<tr>
<td>struct DDS_Endpoint-GroupSeq</td>
<td>required_matched_endpoint_groups</td>
<td>A sequence of endpoint groups, described in Table 7.33 struct DDS_EndpointGroup_t. For Collaborative DataWriters: See Table 7.34 Configuring Collaborative DataWriters with DDS_AvailabilityQosPolicy. For Required Subscriptions: See Table 7.35 Configuring Required Subscriptions with DDS_AvailabilityQosPolicy.</td>
</tr>
</tbody>
</table>
### 7.5.1 AVAILABILITY QosPolicy (DDS Extension)

#### Table 7.33 struct DDS_EndpointGroup_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char*</td>
<td>role_name</td>
<td>Defines the role name of the endpoint group. If used in the AvailabilityQosPolicy on a DataWriter, it specifies the name that identifies a Required Subscription.</td>
</tr>
<tr>
<td>int</td>
<td>quorum_count</td>
<td>Defines the minimum number of members that satisfies the endpoint group. If used in the AvailabilityQosPolicy on a DataWriter, it specifies the number of DataReaders with a specific role name that must acknowledge a DDS sample before the DDS sample is considered to be acknowledged by the Required Subscription.</td>
</tr>
</tbody>
</table>

#### 7.5.1.1 Availability QoS Policy and Collaborative Data Writers

The **Collaborative Data Writers** feature allows you to have multiple Data Writers publishing DDS samples from a common logical data source. The Data Readers will combine the DDS samples coming from the Data Writers in order to reconstruct the correct order at the source. The Availability QosPolicy allows you to configure the DDS sample combination (synchronization) process in the DataReader.

Each DDS sample published in a DDS domain for a given logical data source is uniquely identified by a pair (virtual GUID, virtual sequence number). DDS samples from the same data source (same virtual GUID) can be published by different Data Writers.

A DataReader will deliver a DDS sample (VGUIDn, VSNm) to the application if one of the following conditions is satisfied:

- (GUIDn, SNm-1) has already been delivered to the application.
- All the known Data Writers publishing VGUIDn have announced that they do not have (VGUIDn, VSNm-1).
- None of the known Data Writers publishing VGUIDn have announced potential availability of (VGUIDn, VSNm-1) and both timeouts in this QoS policy have expired.

A DataWriter announces potential availability of DDS samples by using virtual heartbeats. The frequency at which virtual heartbeats are sent is controlled by the protocol parameters virtual_heartbeat_period on page 392 and samples_per_virtual_heartbeat on page 392 (see Table 7.45 DDS_RtpsReliableWriterProtocol_t).

Table 7.34 Configuring Collaborative Data Writers with DDS_AvailabilityQosPolicy describes the fields of this policy when used for a Collaborative Data Writer.

For further information, see Collaborative Data Writers (Chapter 12 on page 727).
### 7.5.1 AVAILABILITY QosPolicy (DDS Extension)

#### Table 7.34 Configuring Collaborative DataWriters with DDS_AvailabilityQosPolicy

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description for Collaborative DataWriters</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_data_availability_waiting_time</td>
<td>Defines how much time to wait before delivering a DDS sample to the application without having received some of the previous DDS samples. A DDS sample identified by (VGUIDn, VSNm) will be delivered to the application if this timeout expires for the DDS sample and the following two conditions are satisfied: None of the known DataWriters publishing VGUID have announced potential availability of (VGUIDn, VSNm-1). The DataWriters for all the endpoint groups specified in required_matched_endpoint_groups on page 370 have been discovered or max_endpoint_availability_waiting_time on the next page has expired.</td>
</tr>
<tr>
<td>max_endpoint_availability_waiting_time</td>
<td>Defines how much time to discover DataWriters providing DDS samples for the same data source. The set of endpoint groups that are required to provide DDS samples for a data source can be configured using required_matched_endpoint_groups on page 370. A non-consecutive DDS sample identified by (GUIDn, SNm) cannot be delivered to the application unless the DataWriters for all the endpoint groups in required_matched_endpoint_groups on page 370 are discovered or this timeout expires.</td>
</tr>
<tr>
<td>required_matched_endpoint_groups</td>
<td>Specifies the set of endpoint groups that are expected to provide DDS samples for the same data source. The quorum count in a group represents the number of DataWriters that must be discovered for that group before the DataReader is allowed to provide non-consecutive DDS samples to the application. A DataWriter becomes a member of an endpoint group by configuring the role_name in the DataWriter’s 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418. The DataWriters created by RTI Persistence Service have a predefined role_name of ‘PERSISTENCE_SERVICE’. For other DataWriters, the role_name is not set by default.</td>
</tr>
</tbody>
</table>

#### 7.5.1.2 Availability QoS Policy and Required Subscriptions

In the context of Required Subscriptions, the Availability QosPolicy can be used to configure a set of required subscriptions on a DataWriter.

Required Subscriptions are preconfigured, named subscriptions that may leave and subsequently rejoin the network from time to time, at the same or different physical locations. Any time a required subscription is disconnected, any DDS samples that would have been delivered to it are stored for delivery if and when the subscription rejoins the network.

Table 7.35 Configuring Required Subscriptions with DDS_AvailabilityQosPolicy describes the fields of this policy when used for a Required Subscription.

For further information, see 7.3.13 Required Subscriptions on page 323.
Table 7.35 Configuring Required Subscriptions with DDS_AvailabilityQosPolicy

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Description for Required Subscriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable_required_subscriptions</td>
<td>Enables support for Required Subscriptions in a DataWriter. If set to TRUE, the 7.5.12 HISTORY QosPolicy on page 420 kind must be KEEP_ALL, because not all samples can be guaranteed to be delivered to the required DataReaders if history kind is KEEP_LAST. Likewise, the 7.5.9 DURABILITY QosPolicy on page 411 writer_depth must be left set to AUTO, because not all samples can be guaranteed to be delivered to the required DataReaders when writer_depth is limited.</td>
</tr>
<tr>
<td>max_data_availability_waiting_time</td>
<td>Not applicable to Required Subscriptions.</td>
</tr>
<tr>
<td>max_endpoint_availability_waiting_time</td>
<td>A sequence of endpoint groups that specify the Required Subscriptions on a DataWriter. Each Required Subscription is specified by a name and a quorum count. The quorum count represents the number of DataReaders that have to acknowledge the DDS sample before it can be considered fully acknowledged for that Required Subscription. A DataReader is associated with a Required Subscription by configuring the role_name in the DataReader's 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
</tbody>
</table>

7.5.1.3 Properties

For DataWriters, all the members in this QosPolicy can be changed after the DataWriter is created except for the member enable_required_subscriptions.

For DataReaders, this QosPolicy cannot be changed after the DataReader is created.

There are no compatibility restrictions for how it is set on the publishing and subscribing sides.

7.5.1.4 Related QosPolicies

- 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418
- 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654
- 7.5.9 DURABILITY QosPolicy on page 411

7.5.1.5 Applicable DDS Entities

- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504
7.5.1.6 System Resource Considerations

The resource limits for the endpoint groups in required_matched_endpoint_groups are determined by two values in the 9.5.4 DOMAIN_PARTICIPANTRESOURCELIMITS QosPolicy (DDS Extension) on page 654:

- max_endpoint_groups
- max_endpoint_group_cumulative_characters

The maximum number of virtual writers (identified by a virtual GUID) that can be managed by a DataReader is determined by the max_remote_virtual_writers in 8.6.2 DATA_READERSOURCELIMITS QosPolicy (DDS Extension) on page 576. When the Subscriber’s access_scope is GROUP, max_remote_virtual_writers determines the maximum number of DataWriter groups supported by the Subscriber. Since the Subscriber may contain more than one DataReader, only the setting of the first applies.

7.5.2 BATCH QosPolicy (DDS Extension)

This QosPolicy can be used to decrease the amount of communication overhead associated with the transmission and (in the case of reliable communication) acknowledgment of small DDS samples, in order to increase throughput.

It specifies and configures the mechanism that allows Connext DDS to collect multiple user data DDS samples to be sent in a single network packet, to take advantage of the efficiency of sending larger packets and thus increase effective throughput.

This QosPolicy can be used to increase effective throughput dramatically for small data DDS samples. Throughput for small DDS samples (size < 2048 bytes) is typically limited by CPU capacity and not by network bandwidth. Batching many smaller DDS samples to be sent in a single large packet will increase network utilization and thus throughput in terms of DDS samples per second.

It contains the members listed in Table 7.36 DDS_BatchQosPolicy.

Table 7.36 DDS_BatchQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>enable</td>
<td>Enables/disables batching.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_data_bytes</td>
<td>Sets the maximum cumulative length of all serialized DDS samples in a batch. Before or when this limit is reached, the batch is automatically flushed. The size does not include the meta-data associated with the batch DDS samples.</td>
</tr>
</tbody>
</table>
### Table 7.36 DDS.BatchQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_samples</td>
<td>Sets the maximum number of DDS samples in a batch. When this limit is reached, the batch is automatically flushed.</td>
</tr>
<tr>
<td>struct DDS_Duration_t</td>
<td>max_flush_delay</td>
<td>Sets the maximum flush delay. When this duration is reached, the batch is automatically flushed. The delay is measured from the time the first DDS sample in the batch is written by the application.</td>
</tr>
<tr>
<td>struct DDS_Duration_t</td>
<td>source_timestamp_resolution</td>
<td>Sets the batch source timestamp resolution. The value of this field determines how the source timestamp is associated with the DDS samples in a batch. A DDS sample written with timestamp 't' inherits the source timestamp 't2' associated with the previous DDS sample, unless (t - t2) is greater than source_timestamp_resolution. If source_timestamp_resolution is DURATION_INFINITE, every DDS sample in the batch will share the source timestamp associated with the first DDS sample. If source_timestamp_resolution is zero, every DDS sample in the batch will contain its own source timestamp corresponding to the moment when the DDS sample was written. The performance of the batching process is better when source_timestamp_resolution is set to DURATION_INFINITE.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>thread_safe_write</td>
<td>Determines whether or not the write operation is thread-safe. If TRUE, multiple threads can call write on the DataWriter concurrently. A setting of FALSE can be used to increase batching throughput for batches with many small DDS samples.</td>
</tr>
</tbody>
</table>

If batching is enabled (not the default), DDS samples are not immediately sent when they are written. Instead, they get collected into a "batch." A batch always contains whole number of DDS samples—a DDS sample will never be fragmented into multiple batches.

A batch is sent on the network ("flushed") when one of the following things happens:

- **User-configurable flushing conditions**
  - A batch size limit (max_data_bytes) is reached.
  - A number of DDS samples are in the batch (max_samples).
  - A time-limit (max_flush_delay) is reached, as measured from the time the first DDS sample in the batch is written by the application.
  - The application explicitly calls a DataWriter's flush() operation.

- **Non-user configurable flushing conditions:**
  - A coherent set starts or ends.
7.5.2 BATCH QosPolicy (DDS Extension)

- The number of DDS samples in the batch is equal to max_samples in RESOURCE_LIMITS for unkeyed topics or max_samples_per_instance in RESOURCE_LIMITS for keyed topics.

Additional batching configuration takes place in the Publisher’s 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345.

The flush() operation is described in 7.3.9 Flushing Batches of DDS Data Samples on page 315.

7.5.2.1 Synchronous and Asynchronous Flushing

Usually, a batch is flushed synchronously:

- When a batch reaches its application-defined size limit (max_data_bytes or max_samples) because the application called write(), the batch is flushed immediately in the context of the writing thread.
- When an application manually flushes a batch, the batch is flushed immediately in the context of the calling thread.
- When the first DDS sample in a coherent set is written, the batch in progress (without including the DDS sample in the coherent set) is immediately flushed in the context of the writing thread.
- When a coherent set ends, the batch in progress is immediately flushed in the context of the calling thread.
- When the number of DDS samples in a batch is equal to max_samples in RESOURCE_LIMITS for unkeyed topics or max_samples_per_instance in RESOURCE_LIMITS for keyed topics, the batch is flushed immediately in the context of the writing thread.

However, some behavior is asynchronous:

- To flush batches based on a time limit (max_flush_delay), enable asynchronous batch flushing in the 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345 of the DataWriter's Publisher. This will cause the Publisher to create an additional thread that will be used to flush batches of that Publisher's DataWriters. This behavior is analogous to the way asynchronous publishing works.
- You may also use batching alongside asynchronous publication with 7.6 FlowControllers (DDS Extension) on page 470. These features are independent of one another. Flushing a batch on an asynchronous DataWriter makes it available for sending to the DataWriter's FlowController. From the point of view of the FlowController, a batch is treated like one large DDS sample.

7.5.2.2 Batching vs. Coalescing

Even when batching is disabled, Connext DDS will sometimes coalesce multiple DDS samples into a single network datagram. For example, DDS samples buffered by a FlowController or sent in response to
a negative acknowledgement (NACK) may be coalesced. This behavior is distinct from DDS sample batching.

DDS samples that are sent individually (not part of a batch) are always treated as separate DDS samples by Connext DDS. Each DDS sample is accompanied by a complete RTPS header on the network (although DDS samples may share UDP and IP headers) and (in the case of reliable communication) a unique physical sequence number that must be positively or negatively acknowledged.

In contrast, batched DDS samples share an RTPS header and an entire batch is acknowledged—positively or negatively—as a unit, potentially reducing the amount of meta-traffic on the network and the amount of processing per individual DDS sample.

Batching can also improve latency relative to simply coalescing. Consider two use cases:

1. A DataWriter is configured to write asynchronously with a FlowController. Even if the FlowController's rules would allow it to publish a new DDS sample immediately, the send will always happen in the context of the asynchronous publishing thread. This context switch can add latency to the send path.

2. A DataWriter is configured to write synchronously but with batching turned on. When the batch is full, it will be sent on the wire immediately, eliminating a thread context switch from the send path.

7.5.2.3 Batching and ContentFilteredTopics

When batching is enabled, content filtering is always done on the reader side.

7.5.2.4 Turbo Mode: Automatically Adjusting the Number of Bytes in a Batch—Experimental Feature

Turbo Mode is an experimental feature that uses an intelligent algorithm that automatically adjusts the number of bytes in a batch at run time according to current system conditions, such as write speed (or write frequency) and DDS sample size. This intelligence is what gives it the ability to increase throughput at high message rates and avoid negatively impacting message latency at low message rates.

To enable Turbo mode, set the DataWriter's property dds.data_writer.enable_turbo_mode to true. Turbo mode is not enabled by default.

Note: If you explicitly enable batching by setting enable to TRUE in BatchQosPolicy, the value of the turbo mode property is ignored and turbo mode is not used.

7.5.2.5 Performance Considerations

The purpose of batching is to increase throughput when writing small DDS samples at a high rate. In such cases, throughput can be increased several-fold, approaching much more closely the physical limitations of the underlying network transport.
However, collecting DDS samples into a batch implies that they are not sent on the network immediately when the application writes them; this can potentially increase latency. However, if the application sends data faster than the network can support, an increased proportion of the network's available bandwidth will be spent on acknowledgements and DDS sample resends. In this case, reducing that overhead by turning on batching could decrease latency while increasing throughput.

As a general rule, to improve batching throughput:

- Set `thread_safe_write` to FALSE when the batch contains a big number of small DDS samples. If you do not use a thread-safe write configuration, asynchronous batch flushing must be disabled.
- Set `source_timestamp_resolution` to `DURATION_INFINITE`. Note that you set this value, every DDS sample in the batch will share the same source timestamp.

Batching affects how often piggyback heartbeats are sent; see `heartbeats_per_max_samples` in Table 7.45 DDS_RtpsReliableWriterProtocol_t.

### 7.5.2.6 Maximum Transport Datagram Size

Batches cannot be fragmented. As a result, the maximum batch size (max_data_bytes) must be set no larger than the maximum transport datagram size. For example, a UDP datagram is limited to 64 KB, so any batches sent over UDP must be less than or equal to that size.

### 7.5.2.7 Bandwidth Considerations

A minimum overhead of 8-bytes is added to each sample in a batch; however, the overhead may be bigger in some cases. For example:

- When you add a source timestamp per sample instead of per batch, there will be 8 more bytes for the source timestamp. You can control this behavior with `writer_qos.batch.source_timestamp_resolution`.
- By default, for keyed topics, Connext DDS adds the key hash for the instance, adding an extra overhead of 20 bytes. If you don’t want to add the key hash and instead get it from the serialized data on the DataReader side, set `writer_qos.protocol.disable_inline_keyhash` to true.
- Disposed/unregistered samples also need an additional 8-byte overhead to mark the status as disposed or unregistered.
- There are other scenarios in which overhead may increase—for example, when using collaborative DataWriters or group order access.

To summarize:
For a data sample for a keyed topic, by default, the overhead will be 32-bytes (8 (minimum) + 20 (for the key hash) + 4 (for the sentinel)). You can reduce this to 8 bytes by not sending the key hash (in which case, the sentinel goes away, too).

For disposed/unregistered samples for a keyed topic, by default, the overhead will be 40-bytes (8 (minimum) + 20 (for the key hash) + 8 (for the status information) + 4 (for the sentinel)). You can reduce this to 20 bytes by not sending the key hash (the sentinel remains for the status information).

For an unkeyed topic, the overhead is typically 8 bytes.

7.5.2.8 Properties

This QosPolicy cannot be modified after the DataWriter is enabled.

Since it is only for DataWriters, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

All batching configuration occurs on the publishing side. A subscribing application does not configure anything specific to receive batched DDS samples, and in many cases, it will be oblivious to whether the DDS samples it processes were received individually or as part of a batch.

Consistency rules:

- **max_samples** must be consistent with **max_data_bytes**: they cannot both be set to LENGTH_UNLIMITED.
- If **max_flush_delay** is not DURATION_INFINITE, **disable_asynchronous_batch** in the 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345 must be FALSE.
- If **thread_safe_write** is FALSE, **source_timestamp_resolution** must be DURATION_INFINITE.

7.5.2.9 Related QosPolicies

To flush batches based on a time limit, enable batching in the 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345 of the DataWriter's Publisher.

Be careful when configuring a DataWriter's 7.5.14 LIFESPAN QoS Policy on page 425 with a duration shorter than the batch flush period (**max_flush_delay**). If the batch does not fill up before the flush period elapses, by default the short duration will cause the DDS samples to be dropped without being sent. (You can, however, change this default behavior. See the last paragraph in this section.)

Do not configure the DataReader's or DataWriter's 7.5.12 HISTORY QosPolicy on page 420 to be shallower than the DataWriter's maximum batch size (**max_samples**). When the HISTORY QosPolicy is shallower on the DataWriter, by default some DDS samples may not be sent. (You can, however, change this default behavior. See the last paragraph in this section.) When the HISTORY QosPolicy is shallower on the DataReader, DDS samples may be lost before being provided to the application.
The initial and maximum numbers of batches that a DataWriter will manage is set in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401.

The maximum number of DDS samples that a DataWriter can store is determined by the value max_samples in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 and max_batches in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401. The limit that is reached first is applied.

The amount of resources required for batching depends on the configuration of the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 and the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401. See 7.5.2.11 System Resource Considerations below.

By default, samples marked as removed in a batch are dropped. Examples of removed samples in a batch are samples that were replaced due to KEEP_LAST_HISTORY_QOS on the DataWriter (see 7.5.12 HISTORY QosPolicy on page 420) or samples that outlived the DataWriter's 7.5.14 LIFESPAN QoS Policy on page 425 duration. You can keep track of the number of these dropped samples via writer_removed_batch_sample_dropped_sample_count in the 8.3.7.2 DATA_READER_CACHE_STATUS on page 517. You can also choose not to drop these samples at all by setting the property dds.data_reader.accept_writer_removed_batch_samples to TRUE (by default it is set to FALSE); you can set this property via the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.

Note: When the DataWriter history depth is shallower than the DataWriter's maximum batch size (max_samples), the excess samples are marked as removed, but you can choose not to drop these removed samples using the accept_writer_removed_batch_samples property. But when theDataReader history depth is shallower than the DataWriter's maximum batch size (max_samples), the excess samples are lost. (These are not affected by the property or included in the dropped sample count.)

7.5.2.10 Applicable DDS Entities

- 7.3 DataWriters on page 287

7.5.2.11 System Resource Considerations

- Batching requires additional resources to store the meta-data associated with the DDS samples in the batch.
  - For unkeyed topics, the meta-data will be at least 8 bytes, with a maximum of 20 bytes.
  - For keyed topics, the meta-data will be at least 8 bytes, with a maximum of 52 bytes.
- Other resource considerations are described in 7.5.2.9 Related QosPolicies on the previous page.

7.5.3 DATA_REPRESENTATION QosPolicy

The DATA_REPRESENTATION QosPolicy is used to configure what form data is represented or expected in on the wire. It indicates which versions (version 1 and version 2) of the Extended Common Data
7.5.3 DATA_REPRESENTATION QosPolicy

Representation (CDR) are offered and requested as well as if and how the data may be compressed, including which compression algorithm is offered and requested.

A DataWriter might also offer a single representation, which indicates the CDR version the DataWriter uses to serialize its data. A DataReader requests one or more representations, which indicate the CDR versions the DataReader accepts. If a DataWriter's offered representation is contained within a reader’s sequence of requested representations, then the offer satisfies the request, and the policies are compatible. Otherwise, they are incompatible. See Table 7.37 DDS_DataRepresentationQosPolicy and 7.5.3.1 Data Representation on the next page for more information.

A DataWriter also offers a single compression_ids value, which is the compression algorithm the DataWriter uses to compress data it sends to matching DataReaders. A DataReader requests zero or more compression algorithms. If a DataWriter offers a compression algorithm that is contained within the algorithms requested by the DataReader, the offer satisfies the request and the policies are compatible. Otherwise, they are incompatible. See Table 7.37 DDS_DataRepresentationQosPolicy and 7.5.3.2 Data Compression on page 383 for more information.

The DATA_REPRESENTATION QosPolicy includes the members in Table 7.37 DDS_DataRepresentationQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

### Table 7.37 DDS_DataRepresentationQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_DataRepresentationIdSeq</td>
<td>value</td>
<td>A sequence of two-byte signed integers corresponding to representation identifiers. The supported identifiers are DDS_XCDR_DATA_REPRESENTATION (Extensible CDR version 1), DDS_XCDR2_DATA_REPRESENTATION (Extensible CDR version 2), and DDS_AUTO_DATA_REPRESENTATION. An empty sequence is equivalent to a sequence with one DDS_XCDR_DATA_REPRESENTATION element. The default value, however, is a sequence with one DDS_AUTO_DATA_REPRESENTATION element. For plain language binding, the value DDS_AUTO_DATA_REPRESENTATION translates to DDS_XCDR_DATA_REPRESENTATION if the @allowed_data_representation annotation either is not specified or contains the value XCDR. Otherwise, it translates to DDS_XCDR2_DATA_REPRESENTATION. For FlatData language binding, DDS_AUTO_DATA_REPRESENTATION translates to XCDR2_DATA_REPRESENTATION. (See 7.5.3.1 Data Representation on the next page for further explanation.) For additional information on the @allowed_data_representation annotation, see Data Representation, in the RTI Connext DDS Core Libraries Extensible Types Guide.</td>
</tr>
</tbody>
</table>
### Table 7.37 DDS_DataRepresentationQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_CompressionSettings_t</td>
<td>compression_settings</td>
<td>Settings related to compressing user data:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• compression_ids: A bitmap that represents the compression algorithm IDs supported by the DataWriter or DataReader. The possible values are: ZLIB, BZIP2, LZ4, MASK_NONE, and MASK_ALL. Only ZLIB is supported if the DataWriter is using both compression and batching. Compression is not supported if also using batching and data protection (via RTI Security Plugins). See 7.5.3.2 Data Compression on the next page. DataWriter creation will fail if more than one algorithm is provided on the DataWriter side (meaning that MASK_ALL is only supported for DataReaderQoS and TopicQoS). Default: MASK_NONE (for DataWriterQoS and TopicQoS), MASK_ALL (for DataReaderQoS)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• writer_compression_level: The level of compression to use when compressing data. The value is a range between 0 and 10. It can be set only for the DataWriterQoS or TopicQoS. A lower compression level results in a faster compression speed, but lower compression ratio. A higher compression level results in a better compression ratio, but slower compression speed. Default: BEST_COMPRESSION (10)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• writer_compression_threshold: The threshold, in bytes, above which a serialized sample is eligible to be compressed. The value is a range between 0 and LENGTH_UNLIMITED. It can be set only for the DataWriterQoS or TopicQoS. Any sample with a serialized size equal to or greater than the threshold will be eligible to be compressed. Only if the compressed size is smaller than the serialized size will the sample be stored and sent compressed on the wire. Setting the threshold to LENGTH_UNLIMITED disables compression. Default: COMPRESSION_THRESHOLD_DEFAULT (8192 bytes). Note: COMPRESSION_THRESHOLD_DEFAULT is not a valid value in XML, it can be set only in code.</td>
</tr>
</tbody>
</table>

See 7.5.3.2 Data Compression on the next page for more details.

#### 7.5.3.1 Data Representation

You can view data representation as a two-step process:

1. As described above, DDS_AUTO_DATA_REPRESENTATION translates to the value DDS_XCDR_DATA_REPRESENTATION or DDS_XCDR2_DATA_REPRESENTATION depending on a few factors. Or you can explicitly set the value to DDS_XCDR_DATA_REPRESENTATION or DDS_XCDR2_DATA_REPRESENTATION. If you let DDS_AUTO_DATA_REPRESENTATION set the value, the following table shows how it will be set, depending on your IDL:
7.5.3 DATA_REPRESENTATION QosPolicy

Table 7.38 How DDS_AUTO_DATA_REPRESENTATION Sets the Value

<table>
<thead>
<tr>
<th>IDL looks like ...</th>
<th>AUTO value translates to ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Struct Point {</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>which is equivalent to:</td>
<td></td>
</tr>
<tr>
<td>@allowed_data_representation(XCDR</td>
<td>XCDR2)</td>
</tr>
<tr>
<td>Struct Point [</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>@allowed_data_representation(XCDR2)</td>
<td></td>
</tr>
<tr>
<td>Struct Point [</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
<tr>
<td>@language_binding(FLAT_DATA)</td>
<td></td>
</tr>
<tr>
<td>Struct PoInt [</td>
<td></td>
</tr>
<tr>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>

2. Once the value is set (either by DDS_AUTO_DATA_REPRESENTATION or explicitly by you), that value determines what the DataWriter writes or the DataReader reads. (Recall that the DataWriter offers one representation; the DataReader requests one or more representations.) The next step is how the DataWriter and DataReader match based on the QoS value. The QoS must be compatible between the DataWriter and the DataReader. The compatible combinations are shown in Table 7.37 DDS_DataRepresentationQosPolicy.

Table 7.39 Valid Reader/Writer Combinations of DataRepresentation

<table>
<thead>
<tr>
<th>DataWriter-offered DataRepresentation value</th>
<th>DataReader-requested DataRepresentation values</th>
</tr>
</thead>
<tbody>
<tr>
<td>XCDR</td>
<td>XCDR</td>
</tr>
<tr>
<td>XCDR</td>
<td>XCDR and XCDR2</td>
</tr>
<tr>
<td>XCDR2</td>
<td>XCDR2</td>
</tr>
<tr>
<td>XCDR2</td>
<td>XCDR and XCDR2</td>
</tr>
</tbody>
</table>

If this QosPolicy is set incompatibly, the ON_OFFERED_INCOMPATIBLE_QOS and ON_REQUESTED_INCOMPATIBLE_QOS statuses will be modified and the corresponding Listeners called for the DataWriter and DataReader respectively.

7.5.3.2 Data Compression

A DataReader with compression enabled can receive samples from DataWriters with or without compression as well as from multiple DataWriters with different compression algorithms. DataWriters cannot optionally send compressed samples to some DataReaders and the same samples, but uncompressed, to other DataReaders that do not support compression.

Table 7.40 Valid Reader/Writer Combinations of Compression IDs shows which DataWriter-s/Data Readers will match depending on their compression IDs:
Table 7.40 Valid Reader/Writer Combinations of Compression IDs

<table>
<thead>
<tr>
<th>DataWriter-offered compression_ids</th>
<th>DataReader-requested compression_ids</th>
<th>NONE</th>
<th>ZLIB</th>
<th>LZ4</th>
<th>BZIP2</th>
<th>MASK_ALL or any combination that includes offered compression_ids</th>
</tr>
</thead>
<tbody>
<tr>
<td>NONE</td>
<td>NONE</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
<td></td>
</tr>
<tr>
<td>ZLIB</td>
<td>incompatible</td>
<td>compatible</td>
<td>incompatible</td>
<td>incompatible</td>
<td>compatible</td>
<td></td>
</tr>
<tr>
<td>LZ4</td>
<td>incompatible</td>
<td>incompatible</td>
<td>compatible</td>
<td>incompatible</td>
<td>compatible</td>
<td></td>
</tr>
<tr>
<td>BZIP2</td>
<td>incompatible</td>
<td>incompatible</td>
<td>incompatible</td>
<td>compatible</td>
<td>compatible</td>
<td></td>
</tr>
</tbody>
</table>

MASK_ALL is not a valid value for the DataWriter, which supports only one compression_ids value.

7.5.3.2.1 compression_ids

You can compare the compression algorithms (LZ4, zlib, and bzip2) by checking their compression ratios against their compression speeds. The compression ratio defines how much the data size is reduced. For example, a ratio of 2 means that the size of the data is reduced by half. The compression speed has a direct impact on the latency of the compressed data; the slower the speed, the higher the latency. Generally, the higher the compression ratio, the lower the speed; the higher the speed, the lower the compression ratio.

Table 7.41 Compression Algorithm References

<table>
<thead>
<tr>
<th>compression_ids</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>MASK_NONE</td>
<td>Default for DataWriterQoS and TopicQoS</td>
</tr>
<tr>
<td>LZ4</td>
<td>See <a href="https://github.com/lz4/lz4">https://github.com/lz4/lz4</a></td>
</tr>
<tr>
<td>ZLIB</td>
<td>See <a href="https://zlib.net/">https://zlib.net/</a></td>
</tr>
<tr>
<td>BZIP2</td>
<td>See <a href="https://www.sourceware.org/bzip2/">https://www.sourceware.org/bzip2/</a></td>
</tr>
<tr>
<td>MASK_ALL</td>
<td>Default for DataReaderQoS</td>
</tr>
</tbody>
</table>

There are many benchmarking resources comparing various compression algorithms. One such resource is https://github.com/inikep/lzbench. LZ4 is considered the fastest of the three builtin algorithms, while zlib and bzip2 give the best compression ratios. Use LZ4 if you want to keep latency as low as possible while maintaining a decent compression ratio. Use zlib or bzip2 if latency is less important in your system than a high compression ratio to reduce bandwidth usage. The choice of which of the three builtin compression algorithms to use depends on the type of data, the rate at which the data is being sent, and latency and bandwidth considerations. It is a good idea for you to understand the strengths and weaknesses of each of
the builtin algorithms, and perform benchmarking in your own system so that you can choose the algorithm that is best suited to your system.

When you specify compression settings for a Topic, all DataWriters and DataReaders for that Topic inherit the Topic's compression settings. If you specify multiple compression algorithms for a Topic, the DataReader will use all of them, but since the DataWriter can have only one algorithm enabled, it will choose one of them, in the following order: ZLIB, BZIP2, and LZ4.

**Notes:**

- When the `serialize_key_with_dispose` field in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 is enabled and a dispose message is sent, the serialized key is not compressed.

- The only algorithm supported when compression and batching are enabled on the same DataWriter is ZLIB, because zlib is the only builtin algorithm that supports stream-based compression with acceptable performance. Stream-based compression allows Connext DDS to compress and build the batch as samples are written into the batch. (LZ4 also supports stream-based compression, but with a high performance penalty, so RTI has decided not to support this mode in Connext DDS.)

- **If you are using Security Plugins and batching:** The combination of compression, batching, and data protection (via Security Plugins) is not supported and will result in a DataWriter creation error. Consider using RTPS protection instead of data protection if compression and batching are also required. The reason for this restriction is that when using batching, compression is applied to the entire batch, while data protection is applied to each of the batch samples individually. Compressing already encrypted data results in an expansion of the data instead of a reduction of it because encrypted data does not lend itself to compression.

### 7.5.3.2.2 **writer_compression_level**

Each level between 0 and 10 has trade-offs between compression ratio and compression speed, with 1 representing the fastest speed and lowest compression ratio and 10 representing the slowest speed and highest compression ratio. (0 disables compression.)

Connext DDS also provides the following `writer_compression_level` values:

- **BEST_COMPRESSION.** This value is the same as 10. With this value, Connext DDS chooses the best compression level for the given algorithm.
- **BEST_SPEED.** This value is the same as 1. With this value, Connext DDS chooses the fastest compression speed for whatever algorithm is chosen.

BEST_COMPRESSION and BEST_SPEED do not vary dynamically depending on the algorithm and the size of the data. They have a strict one-to-one mapping to the algorithms' compression ratios/speeds as follows:
7.5.3 DATA_REPRESENTATION QosPolicy

- **zlib**

<table>
<thead>
<tr>
<th>writer_compression_level</th>
<th>zlib mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEST_COMPRESSION = 10</td>
<td>level = 9</td>
</tr>
<tr>
<td>BEST_SPEED = 1</td>
<td>level = 1</td>
</tr>
</tbody>
</table>

For the rest of the values, a linear normalization is applied, so any `writer_compression_level` value you enter in the range of 1 to 10 is translated to the range used by ZLIB between 1 and 9. See the zlib documentation for the `compress2` function for more details on how the level parameter is used.

- **LZ4**

<table>
<thead>
<tr>
<th>writer_compression_level</th>
<th>LZ4 mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEST_COMPRESSION = 10</td>
<td>acceleration = 0</td>
</tr>
<tr>
<td>BEST_SPEED = 1</td>
<td>acceleration = 30</td>
</tr>
</tbody>
</table>

For the rest of the values, a linear normalization is applied, so any `writer_compression_level` value you enter in the range of 1 to 10 is translated to the range used by LZ4 between 30 and 0. Although technically the acceleration value is unbounded, Connext DDS sets the limit at 30; beyond that, no compression occurs in most cases. See the LZ4 documentation for the `LZ4_compress_fast` function for more details on how the acceleration parameter is used.

- **bzip2**

<table>
<thead>
<tr>
<th>writer_compression_level</th>
<th>bzip2 mapped value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEST_COMPRESSION = 10</td>
<td>blockSize100k = 9</td>
</tr>
<tr>
<td>BEST_SPEED = 1</td>
<td>blockSize100k = 1</td>
</tr>
</tbody>
</table>

For the rest of the values, a linear normalization is applied, so any `writer_compression_level` value you enter in the range of 1 to 10 is translated to the range used by bzip2 between 1 and 9. See the bzip2 documentation for the `BZ2_bzBuffToBuffCompress` function for more details on how the blockSize100k parameter is used.

**7.5.3.2.3 writer_compression_threshold**

Any sample with a serialized size equal to or greater than this threshold (see Table 7.37 DDS_DataRepresentationQosPolicy) is eligible to be compressed.
There are two scenarios where a sample, even with compression enabled on the DataWriter, is not compressed:

- Any sample with a serialized size lower than the writer_compression_threshold will not be compressed.

  If batching is enabled: a batch will not be compressed if the maximum serialized size of the batch \((\text{max\_sample\_serialized\_size} \times \text{max\_samples in the batch})\) is smaller than the writer_compression_threshold. See information about \text{max\_samples} in 7.5.2 BATCH QosPolicy (DDS Extension) on page 374.

- If the compressed size is bigger than the sample's serialized size, the compressed sample will be discarded and the original sample will be sent instead.

7.5.3.2.4 Connext DDS Micro

Connext DDS Micro does not interoperate with DataWriters that send compressed data.

7.5.3.2.5 Performance Considerations when Using Content Filtering and Compression

Samples are stored compressed in the DataWriter's queue. When a sample is being written and there are matching DataReaders using ContentFilteredTopics, the DataWriter will apply the filter and then compress the sample. In some cases, a sample needs to be filtered again after it has already been compressed. This can happen, for example, when a non-VOLATILE, late-joining DataReader with a ContentFilteredTopic is discovered by the DataWriter or a TopicQuery is issued by an existing DataReader. If a filtering operation occurs on the DataWriter side after the sample is already compressed, the sample must be decompressed to apply the filter, increasing the latency for these requested samples. Note that in these scenarios the original compressed sample is kept around, so a sample is never compressed twice. In other words, Connext DDS decompresses the sample into a separate buffer, performs the filtering, and then either sends or doesn't send the compressed sample.

7.5.3.2.6 Using Compression with FlatData language binding and Zero Copy Transfer over Shared Memory

See FlatData's section 23.1.4.3.1 Interactions with RTI Security Plugins and Compression on page 961 for notes about interactions with the FlatData language binding.

See Zero Copy's section 23.1.5.1.5 Interactions with RTI Security Plugins and Compression on page 968 for information about interactions with Zero Copy transfer over shared memory.

7.5.3.3 Properties

This QosPolicy cannot be modified after the Entity has been enabled.
7.5.3.4 Applicable Entities

- 5.1 Topics on page 215
- 7.3 Data Writers on page 287
- 8.3 Data Readers on page 504

7.5.4 DATATAG QosPolicy

The DATATAG QosPolicy can be used to associate a set of tags in the form of (name, value) pairs with a DataReader or DataWriter. The Access Control plugin may use the tags to determine publish and subscribe permissions.

The DATATAG QosPolicy is similar to the PropertyQosPolicy, except you cannot select whether or not a particular pair should be propagated (included in the built-in topic); data tags are always propagated. For example, with the Property QoS, it would be possible for a DomainParticipant to allow its own endpoint (DataReader or DataWriter) based on some required properties, and for a remote DomainParticipant to deny that same endpoint because the endpoint chose not to propagate the required properties to the remote DomainParticipant. To avoid such inconsistencies—and because other participants must know about a DomainParticipant’s security credentials—data tags in the DATATAG QosPolicy are always propagated.

This policy includes the member listed in Table 7.42 DDS_DataTagQosPolicy.

Table 7.42 DDS_DataTagQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TagSeq</td>
<td>tags</td>
<td>A sequence of (name, value) string pairs.</td>
</tr>
</tbody>
</table>

You can manipulate the sequence of tags (name, value pairs) with the standard methods available for sequences. You can also use the helper class, DataTagQosPolicyHelper, which provides another way to work with a DataTagQosPolicy object. The DataTagQosPolicyHelper operations are described in the following table. For more information, see the API Reference HTML documentation.

Table 7.43 DDS_DataTagQosPolicyHelper Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_number_of_tags</td>
<td>Gets the number of data tags in the input policy.</td>
</tr>
<tr>
<td>assert_tag</td>
<td>Asserts the data tag identified by name in the input policy. (Either adds it, or replaces an existing one.)</td>
</tr>
<tr>
<td>add_tag</td>
<td>Adds a new data tag to the input policy.</td>
</tr>
<tr>
<td>lookup_tag</td>
<td>Searches for a data tag in the input policy given its name.</td>
</tr>
<tr>
<td>remove_tag</td>
<td>Removes a data tag from the input policy.</td>
</tr>
</tbody>
</table>
7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)

7.5.4.1 Properties

This QosPolicy cannot be modified after the Entity has been created. There is no requirement that the publishing and subscribing sides use compatible values.

7.5.4.2 Related QosPolicies

9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654

7.5.4.3 Applicable Entities

- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.4.4 System Resource Considerations

9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654 contains several fields for configuring the resources associated with the data tags stored in this QosPolicy.

7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)

Connext DDS uses a standard protocol for packet (user and meta data) exchange between applications. The DataWriterProtocol QosPolicy gives you control over configurable portions of the protocol, including the configuration of the reliable data delivery mechanism of the protocol on a per DataWriter basis.

These configuration parameters control timing and timeouts, and give you the ability to trade off between speed of data loss detection and repair, versus network and CPU bandwidth used to maintain reliability.

It is important to tune the reliability protocol on a per DataWriter basis to meet the requirements of the end-user application so that data can be sent between DataWriters and DataReaders in an efficient and optimal manner in the presence of data loss. You can also use this QosPolicy to control how Connext DDS responds to "slow" reliable DataReaders or ones that disconnect or are otherwise lost.

This policy includes the members presented in Table 7.44 DDS_DataWriterProtocolQosPolicy and Table 7.45 DDS_RtpsReliableWriterProtocol_t. For defaults and valid ranges, please refer to the API Reference HTML documentation.

For details on the reliability protocol used by Connext DDS, see Reliable Communications (Chapter 11 on page 688). See the 7.5.21 RELIABILITY QosPolicy on page 447 for more information on per-DataReader/DataWriter reliability configuration. The 7.5.12 HISTORY QosPolicy on page 420 and 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 also play important roles in the DDS reliability protocol.
### Table 7.44 DDS_DataWriterProtocolQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| DDS_GUID_t         | virtual_guid        | The virtual GUID (Global Unique Identifier) is used to uniquely identify the same DataWriter across multiple incarnations. In other words, this value allows Connext DDS to remember information about a DataWriter that may be deleted and then recreated.  
Connext DDS uses the virtual GUID to associate a durable writer history to a DataWriter.  
Persistence Service uses the virtual GUID to send DDS samples on behalf of the original DataWriter.  
A DataReader persists its state based on the virtual GUIDs of matching remote DataWriters.  
For more information, see 13.2 Durability and Persistence Based on Virtual GUIDs on page 737. By default, Connext DDS will assign a virtual GUID automatically. If you want to restore the state of the durable writer history after a restart, you can retrieve the value of the writer’s virtual GUID using the DataWriter’s get_qos() operation, and set the virtual GUID of the restarted DataWriter to the same value. |
| DDS_Unsigned-Long  | rtps_object_id      | Determines the DataWriter’s RTPS object ID, according to the DDS-RTPS Interoperability Wire Protocol. Only the last 3 bytes are used; the most significant byte is ignored.  
The rtps_host_id, rtps_app_id, and rtps_instance_id in the 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670, together with the 3 least significant bytes in rtps_object_id, and another byte assigned by Connext DDS to identify the entity type, forms the BuiltinTopicKey in PublicationBuiltinTopicData. |
| DDS_Boolean        | push_on_write       | Controls when a DDS sample is sent after write() is called on a DataWriter. If TRUE, the DDS sample is sent immediately; if FALSE, the DDS sample is put in a queue until an ACK/NACK is received from a reliable DataReader. |
| DDS_Boolean        | disable_positive_acks | Determines whether matching DataReaders send positive acknowledgements (ACKs) to the DataWriter. When TRUE, the DataWriter will keep DDS samples in its queue for ACK-disabled readers for a minimum keep duration (see 7.5.3 Disabling Positive Acknowledgements on page 396). When strict reliability is not required, setting this to TRUE reduces overhead network traffic. |
| DDS_Boolean        | disable_inline_keyhash | Controls whether or not the key-hash is propagated on the wire with DDS samples. This field only applies to keyed writers.  
Connext DDS associates a key-hash (an internal 16-byte representation) with each key.  
When FALSE, the key-hash is sent on the wire with every data instance.  
When TRUE, the key-hash is not sent on the wire (so the readers must compute the value using the received data).  
If the reader is CPU bound, sending the key-hash on the wire may increase performance, because the reader does not have to get the key-hash from the data. If the writer is CPU bound, sending the key-hash on the wire may decrease performance, because it requires more bandwidth (16 more bytes per DDS sample).  
Setting disable_inline_keyhash to TRUE is not compatible with using RTI Database Integration Service or RTI Recording Service. |
### Table 7.44 DDS_DataWriterProtocolQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>serialize_key_with_dispose</td>
<td>Controls whether or not the serialized key is propagated on the wire with dispose notifications. This field only applies to keyed writers. RTI recommends setting this field to TRUE if there are DataReaders with propagate_dispose_of_unregistered_instances (in the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570) also set to TRUE. Important: When this field TRUE, batching will not be compatible with RTI Data Distribution Service 4.3e, 4.4b, or 4.4c—the DataReaders will receive incorrect data and/or encounter deserialization errors.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>propagate_app_ack_with_no_response</td>
<td>Controls whether or not a DataWriter receives on_application_acknowledgment() notifications with an empty or invalid response. When FALSE, on_application_acknowledgment() will not be invoked if the DDS sample being acknowledged has an empty or invalid response.</td>
</tr>
<tr>
<td>DDS_RtpsReliableWriterProtocol_t</td>
<td>rtps_reliable_writer</td>
<td>This structure includes the fields in Table 7.45 DDS_RtpsReliableWriterProtocol_t.</td>
</tr>
<tr>
<td>DDS_Sequence_t</td>
<td>initial_virtual_sequence_number</td>
<td>Determines the initial virtual sequence number for this DataWriter. By default, the virtual sequence number of the first sample published by a DataWriter is 1 for DataWriters that do not use durable writer history. For durable writers, the default virtual sequence number is the last sequence number they published in a previous execution, plus one. So, when a non-durable DataWriter is restarted and must continue communicating with the same DataReaders, its samples start over with sequence number 1. Durable DataWriters start over where the last sequence number left off, plus one. This QoS setting allows overwriting the default initial virtual sequence number. Normally, this parameter is not expected to be modified; however, in some scenarios when continuing communication after restarting, applications may require the DataWriter's virtual sequence number to start at something other than the value described above. An example would be to enable non-durable DataWriters to start at the last sequence number published, plus one, similar to the durable DataWriter. This property enables you to make such a configuration, if desired. The virtual sequence number can be overwritten as well on a per sample basis by updating DDS_WriteParams_t::identity in FooDataWriter_write_w_params.</td>
</tr>
</tbody>
</table>

### Table 7.45 DDS_RtpsReliableWriterProtocol_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>low_watermark</td>
<td>Queue levels that control when to switch between the regular and fast heartbeat rates (heartbeat_period on the next page and fast_heartbeat_period on the next page). See 7.5.5.1 High and Low Watermarks on page 394.</td>
</tr>
<tr>
<td></td>
<td>high_watermark</td>
<td></td>
</tr>
</tbody>
</table>
### Table 7.45 DDS_RtpsReliableWriterProtocol_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>heartbeat_period</td>
<td>Rates at which to send heartbeats to DataReaders with unacknowledged DDS samples. See 7.5.5.2 Normal, Fast, and Late-Joiner Heartbeat Periods on page 395 and 11.3.4.1 How Often Heartbeats are Resent (heartbeat_period) on page 704.</td>
</tr>
<tr>
<td></td>
<td>fast_heartbeat_period</td>
<td></td>
</tr>
<tr>
<td></td>
<td>late_joiner_heartbeat_period</td>
<td></td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>virtual_heartbeat_period</td>
<td>The rate at which a reliable DataWriter will send virtual heartbeats. Virtual heartbeat informs the reliable DataReader about the range of DDS samples currently present for each virtual GUID in the reliable writer's queue. See 7.5.5.6 Virtual Heartbeats on page 399.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>samples_per_virtual_heartbeat</td>
<td>The number of DDS samples that a reliable DataWriter must publish before sending a virtual heartbeat. See 7.5.5.6 Virtual Heartbeats on page 399.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_heartbeat_retries</td>
<td>Maximum number of periodic heartbeats sent without receiving an ACK/NACK packet before marking a DataReader 'inactive.' When a DataReader has not acknowledged all the DDS samples the reliable DataWriter has sent to it, and max_heartbeat_retries number of periodic heartbeats have been sent without receiving any ACK/NACK packets in return, the DataReader will be marked as inactive (not alive) and be ignored until it resumes sending ACK/NACKs. Note that piggyback heartbeats do not count towards this value. See 11.3.4.4 Controlling How Many Times Heartbeats are Resent (max_heartbeat_retries) on page 709.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>inactivate_non_progressing_readers</td>
<td>Allows the DataWriter to treat DataReaders that send successive non-progressing NACK packets as inactive. See 11.3.4.5 Treating Non-Progressing Readers as Inactive Readers (inactivate_nonprogressing_readers) on page 709.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>heartbeats_per_max_samples</td>
<td>When a DataWriter is configured with a fixed send window size (min_send_window_size is equal to effective max_send_window_size), a piggyback heartbeat is sent every [effective max_send_window_size/heartbeats_per_max_samples] number of samples written. (See 7.5.5.4 Configuring the Send Window Size on page 397.) Otherwise, the number of piggyback heartbeats sent is scaled according to the current size of the send window. For example, consider a heartbeats_per_max_samples of 50. If the current send window size is 100, a piggyback heartbeat will be sent every two samples. If the send window size grows to 150, a piggyback heartbeat will be sent every three samples, and so on. Additionally, when the send window size grows, a piggyback heartbeat is sent with the next sample. (If it weren’t, the sending of that heartbeat could be delayed, since the heartbeat rate scales with the increasing window size.) The effective max send window is calculated as follows: Without batching, it is the minimum of max_samples in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 or max_send_window_size. With batching, it is the minimum of max_batches in the 7.5.6 DATA_WRITER RESOURCE_LIMITS QosPolicy (DDS Extension) on page 451 or max_send_window_size. If heartbeats_per_max_samples is set to zero, no piggyback heartbeat will be sent. If the current send window size is LENGTH_UNLIMITED, 100 million is assumed as the effective max send window.</td>
</tr>
</tbody>
</table>
### Table 7.45 DDS_RtpsReliableWriterProtocol_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>disable_repair_piggyback_heartbeat</td>
<td>When samples are repaired, the DataWriter resends the number of bytes indicated in max_bytes_per_nack_response and a piggyback heartbeat with each message. You can configure the DataWriter to not send the piggyback heartbeat, by setting this field to TRUE, and instead rely on the late_joiner_heartbeat_period to control the throughput used to repair samples. This field is only mutable for the DataWriter QoS and not for the Discovery Config QoS of the DomainParticipant.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>min_nack_response_delay</td>
<td>Minimum delay to respond to an ACK/NACK. When a reliable DataWriter receives an ACK/NACK from a DataReader, the DataWriter can choose to delay a while before it sends repair DDS samples or a heartbeat. This set the value of the minimum delay. See 11.3.4.6 Coping with Redundant Requests for Missing DDS Samples (max_nack_response_delay) on page 709.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>max_nack_response_delay</td>
<td>Maximum delay to respond to a ACK/NACK. This sets the value of maximum delay between receiving an ACK/NACK and sending repair DDS samples or a heartbeat. A longer wait can help prevent storms of repair packets if many DataReaders send NACKs at the same time. However, it delays the repair, and hence increases the latency of the communication. See 11.3.4.6 Coping with Redundant Requests for Missing DDS Samples (max_nack_response_delay) on page 709.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>nack_suppression_duration</td>
<td>How long consecutive NACKs are suppressed. When a reliable DataWriter receives consecutive NACKs within a short duration, this may trigger the DataWriter to send redundant repair messages. This value sets the duration during which consecutive NACKs are ignored, thus preventing redundant repairs from being sent.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_bytes_per_nack_response</td>
<td>Maximum bytes in a repair package. When a reliable DataWriter resends DDS samples, the total package size is limited to this value. Note: The reliable DataWriter will always send at least one sample. See 11.3.4.3 Controlling Packet Size for Resent DDS Samples (max_bytes_per_nack_response) on page 708.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>disable_positive_acks_min_sample_keep_duration</td>
<td>Minimum duration that a DDS sample will be kept in the DataWriter's queue for ACK-disabled DataReaders. See 7.5.5.3 Disabling Positive Acknowledgements on page 396 and 11.3.4.7 Disabling Positive Acknowledgements (disable_positive_acks_min_sample_keep_duration) on page 711.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>disable_positive_acks_max_sample_keep_duration</td>
<td>Maximum duration that a DDS sample will be kept in the DataWriter's queue for ACK-disabled readers.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>disable_positive_acks_enable_adaptive_sample_keep_duration</td>
<td>Enables automatic dynamic adjustment of the 'keep duration' in response to network congestion.</td>
</tr>
</tbody>
</table>
Table 7.45 DDS_RtpsReliableWriterProtocol_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>disable_positive_acls_increase_sample_keep_duration_factor</td>
<td>When the ‘keep duration’ is dynamically controlled, the lengthening of the ‘keep duration’ is controlled by this factor, which is expressed as a percentage. When the adaptive algorithm determines that the keep duration should be increased, this factor is multiplied with the current keep duration to get the new longer keep duration. For example, if the current keep duration is 20 milliseconds, using the default factor of 150% would result in a new keep duration of 30 milliseconds.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>disable_positive_acls_decrease_sample_keep_duration_factor</td>
<td>When the ‘keep duration’ is dynamically controlled, the shortening of the ‘keep duration’ is controlled by this factor, which is expressed as a percentage. When the adaptive algorithm determines that the keep duration should be decreased, this factor is multiplied with the current keep duration to get the new shorter keep duration. For example, if the current keep duration is 20 milliseconds, using the default factor of 95% would result in a new keep duration of 19 milliseconds.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>min_send_window_size</td>
<td>Minimum and maximum size for the window of outstanding DDS samples.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_send_window_size</td>
<td>See 7.5.5.4 Configuring the Send Window Size on page 397.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>send_window_decrease_factor</td>
<td>Scales the current send-window size down by this percentage to decrease the effective send-rate in response to received negative acknowledgement. See 7.5.5.4 Configuring the Send Window Size on page 397.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>enable_multicast_periodic_heartbeat</td>
<td>Controls whether or not periodic heartbeat messages are sent over multicast. When enabled, if a reader has a multicast destination, the writer will send its periodic HEARTBEAT messages to that destination. Otherwise, if not enabled or the reader does not have a multicast destination, the writer will send its periodic HEARTBEATs over unicast.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>multicast_resend_threshold</td>
<td>Sets the minimum number of requesting readers needed to trigger a multicast resend. See 7.5.5.7 Resending Over Multicast on page 399.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>send_window_increase_factor</td>
<td>Scales the current send-window size up by this percentage to increase the effective send-rate when a duration has passed without any received negative acknowledgements. See 7.5.5.4 Configuring the Send Window Size on page 397</td>
</tr>
<tr>
<td>DDS_Duration</td>
<td>send_window_update_period</td>
<td>Period in which DataWriter checks for received negative acknowledgements and conditionally increases the send-window size when none are received. See 7.5.5.4 Configuring the Send Window Size on page 397</td>
</tr>
</tbody>
</table>

7.5.5.1 High and Low Watermarks

When the number of unacknowledged DDS samples in the current send-window of a reliable DataWriter meets or exceeds high_watermark on page 391, the 7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305 will be changed appropriately, a listener callback will be triggered, and the DataWriter will start heartbeating its matched DataReaders at fast_heartbeat_period on page 392.
When the number of DDS samples meets or falls below low_watermark on page 391, the 7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305 will be changed appropriately, a listener callback will be triggered, and the heartbeat rate will return to the "normal" rate (heartbeat_period on page 392).

Having both high and low watermarks (instead of one) helps prevent rapid flickering between the rates, which could happen if the number of DDS samples hovers near the cut-off point.

Increasing the high and low watermarks will make the DataWriters less aggressive about seeking acknowledgments for sent data, decreasing the size of traffic spikes but slowing performance.

Decreasing the watermarks will make the DataWriters more aggressive, increasing both network utilization and performance.

If batching is used, high_watermark on page 391 and low_watermark on page 391 refer to batches, not DDS samples.

When min_send_window_size on the previous page and max_send_window_size on the previous page are not equal, the low and high watermarks are scaled down linearly to stay within the current send-window size. The value provided by configuration corresponds to the high and low watermarks for the max_send_window_size on the previous page.

### 7.5.5.2 Normal, Fast, and Late-Joiner Heartbeat Periods

The normal heartbeat_period on page 392 is used until the number of DDS samples in the reliable DataWriter's queue meets or exceeds high_watermark on page 391; then fast_heartbeat_period on page 392 is used. Once the number of DDS samples meets or drops below low_watermark on page 391, the normal rate (heartbeat_period on page 392) is used again.

- fast_heartbeat_period on page 392 must be <= heartbeat_period on page 392

Increasing fast_heartbeat_period on page 392 increases the speed of discovery, but results in a larger surge of traffic when the DataWriter is waiting for acknowledgments.

Decreasing heartbeat_period on page 392 decreases the steady state traffic on the wire, but may increase latency by decreasing the speed of repairs for lost packets when the writer does not have very many outstanding unacknowledged DDS samples.

Having two periodic heartbeat rates, and switching between them based on watermarks:

- Ensures that all DataReaders receive all their data as quickly as possible (the sooner they receive a heartbeat, the sooner they can send a NACK, and the sooner the DataWriter can send repair DDS samples);
• Helps prevent the DataWriter from overflowing its resource limits (as its queue starts the fill, the DataWriter sends heartbeats faster, prompting the DataReaders to acknowledge sooner, allowing the DataWriter to purge these acknowledged DDS samples from its queue);

• Tunes the amount of network traffic. (Heartbeats and NACKs use up network bandwidth like any other traffic; decreasing the heartbeat rates, or increasing the threshold before the fast rate starts, can smooth network traffic—at the expense of discovery performance).

The late_joiner_heartbeat_period on page 392 is used when a reliable DataReader joins after a reliable DataWriter (with non-volatile Durability) has begun publishing DDS samples. Once the late-joining DataReader has received all cached DDS samples, it will be serviced at the same rate as other reliable DataReaders.

• late_joiner_heartbeat_period on page 392 must be <= heartbeat_period on page 392

7.5.5.3 Disabling Positive Acknowledgements

When strict reliable communication is not required, you can configure Connext DDS so that it does not send positive acknowledgements (ACKs). In this case, reliability is maintained solely based on negative acknowledgements (NACKs). The removal of ACK traffic may improve middleware performance. For example, when sending DDS samples over multicast, ACK-storms that previously may have hindered DataWriters and consumed overhead network bandwidth are now precluded.

By default, DataWriters and DataReaders are configured with positive ACKS enabled. To disable ACKs, either:

• Configure the DataWriter to disable positive ACKs for all matching DataReaders (by setting disable_positive_acks to TRUE in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389).

• Disable ACKs for individual DataReaders (by setting disable_positive_acks to TRUE in the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570).

If ACKs are disabled, instead of the DataWriter holding a DDS sample in its send queue until all of its DataReaders have ACKed it, the DataWriter will hold a DDS sample for a configurable duration. This "keep-duration" starts when a DDS sample is written. When this time elapses, the DDS sample is logically considered as acknowledged by its ACK-disabled readers.

The length of the "keep-duration" can be static or dynamic, depending on how rtps_reliable_writer.disable_positive_acks_enable_adaptive_sample_keep_duration is set.

• When the length is static, the "keep-duration" is set to the minimum (rtps_reliable_writer.disable_positive_acks_min_sample_keep_duration).
When the length is dynamic, the "keep-duration" is dynamically adjusted between the minimum and maximum durations (rtps_reliable_writer.disable_positive_ack_min_sample_keep_duration and rtps_reliable_writer.disable_positive_ack_max_sample_keep_duration).

Dynamic adjustment maximizes throughput and reliability in response to current network conditions: when the network is congested, durations are increased to decrease the effective send rate and relieve the congestion; when the network is not congested, durations are decreased to increase the send rate and maximize throughput.

You should configure the minimum "keep-duration" to allow at least enough time for a possible NACK to be received and processed. When a DataWriter has both matching ACK-disabled and ACK-enabled DataReaders, it holds a DDS sample in its queue until all ACK-enabled DataReaders have ACKed it and the "keep-duration" has elapsed.

See also: 11.3.4.7 Disabling Positive Acknowledgements (disable_positive_acks_min_sample_keep_duration) on page 711.

7.5.5.4 Configuring the Send Window Size

When a reliable DataWriter writes a DDS sample, it keeps the DDS sample in its queue until it has received acknowledgements from all of its subscribing DataReaders. The number of these outstanding DDS samples is referred to as the DataWriter's "send window." Once the number of outstanding DDS samples has reached the send window size, subsequent writes will block until an outstanding DDS sample is acknowledged.

Configuration of the send window sets a minimum and maximum size, which may be unlimited. The send window size is initialized to the minimum size. The min and max send windows can be the same. When set differently, the send window will dynamically change in response to detected network congestion, as signaled by received negative acknowledgements. When NACKs are received, the DataWriter responds to the slowed reader by decreasing the send window by the send_window_decrease_factor to throttle down its effective send rate. The send window will not be decreased to less than the min_send_window_size. After a period (send_window_update_period) during which no NACKs are received, indicating that the reader is catching up, the DataWriter will increase the send window size to increase the effective send rate by the percentage specified by send_window_increase_factor. The send window will increase to no greater than the max_send_window_size.

When both min_send_window_size and max_send_window_size are unlimited, either the resource limits max_samples in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 (for non-batching) or max_batches in 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401 (for batching) serves as the effective max_send_window_size.

When either max_samples (for non-batching) or max_batches (for batching) is less than max_send_window_size, it serves as the effective max_send_window_size. If it is also less than min_send_window_size, then effectively both min and max send-window sizes are equal to max_samples or max_batches.
### 7.5.5 Propagating Serialized Keys with Disposed-Instance Notifications

This section describes the interaction between these two fields:

- `serialize_key_withDispose` in 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389
- `propagate_dispose_of_unregistered_instances` in 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570

RTI recommends setting `serialize_key_withDispose` to TRUE if there are DataReaders with `propagate_dispose_of_unregistered_instances` also set to TRUE. However, it is permissible to set one to TRUE and the other to FALSE. The following examples will help you understand how these fields work.

See also: 7.3.14.3 Disposing Instances on page 328.

**Example 1**

1. *DataWriter’s `serialize_key_withDispose` = FALSE*
2. *DataReader’s `propagate_dispose_of_unregistered_instances` = TRUE*
3. *DataWriter calls `dispose()` before writing any DDS samples*
4. *DataReader calls `take()` and receives a disposed-instance notification (without a key)*
5. *DataReader calls `get_key_value()`, which returns an error because there is no key associated with the disposed-instance notification*

**Example 2**

1. *DataWriter’s `serialize_key_withDispose` = TRUE*
2. *DataReader’s `propagate_dispose_of_unregistered_instances` = FALSE*
3. *DataWriter calls `dispose()` before writing any DDS samples*
4. *DataReader calls `take()`, which does not return any DDS samples because none were written, and it does not receive any disposed-instance notifications because `propagate_dispose_of_unregistered_instances` = FALSE*

**Example 3**

1. *DataWriter’s `serialize_key_withDispose` = TRUE*
2. *DataReader’s `propagate_dispose_of_unregistered_instances` = TRUE*
3. *DataWriter calls `dispose()` before writing any DDS samples*
4. `DataReader` calls `take()` and receives the disposed-instance notification
5. `DataReader` calls `get_key_value()` and receives the key for the disposed-instance notification

**Example 4**

1. `DataWriter`'s `serialize_key_with_dispose` = TRUE
2. `DataReader`'s `propagate_dispose_of_unregistered_instances` = TRUE
3. `DataWriter` calls `write()`, which writes a DDS sample with a key
4. `DataWriter` calls `dispose()`, which writes a disposed-instance notification with a key
5. `DataReader` calls `take()` and receives a DDS sample and a disposed-instance notification; both have keys
6. `DataReader` calls `get_key_value()` with no errors

**Note:** Persistence Service `DataReaders` ignore the serialized key propagated with dispose updates. Persistence Service `DataWriters` cannot propagate the serialized key with dispose, and therefore ignore the `serialize_key_with_dispose` setting on the `DataWriter` QoS.

**7.5.5.6 Virtual Heartbeats**

Virtual heartbeats announce the availability of DDS samples with the Collaborative DataWriters feature described in 8.6.1 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 570, where multiple `DataWriters` publish DDS samples from a common logical data-source (identified by a virtual GUID).

When 7.4.6 PRESENTATION QosPolicy on page 362 `access_scope` is set to TOPIC or INSTANCE on the `Publisher`, the virtual heartbeat contains information about the DDS samples contained in the `DataWriter` queue.

When presentation `access_scope` is set to GROUP on the `Publisher`, the virtual heartbeat contains information about the DDS samples in the queues of all `DataWriters` that belong to the `Publisher`.

**7.5.5.7 Resending Over Multicast**

Given `DataReaders` with multicast destinations, when a `DataReader` sends a NACK to request for DDS samples to be resent, the `DataWriter` can either resend them over unicast or multicast. Though resending over multicast would save bandwidth and processing for the `DataWriter`, the potential problem is that there could be `DataReaders` of the multicast group that did not request for any resends, yet they would have to process, and drop, the resent DDS samples.

Thus, to make each multicast resend more efficient, the `multicast_resend_threshold` is set as the minimum number of `DataReaders` of the same multicast group that the `DataWriter` must receive NACKs from.
within a single response-delay duration. This allows the DataWriter to coalesce near-simultaneous unicast resends into a multicast resend, and it allows a "vote" from DataReaders of a multicast group to exceed a threshold before resending over multicast.

The multicast_resend_threshold must be set to a positive value. Note that a threshold of 1 means that all resends will be sent over multicast. Also, note that a DataWriter with a zero NACK response-delay (i.e., both min_nack_response_delay and max_nack_response_delay are zero) will resend over multicast only if the threshold is 1.

7.5.5.8 Example

For information on how to use the fields in Table 7.45 DDS_RtpsReliableWriterProtocol_t, see 11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy on page 704.

The following describes a use case for when to change push_on_write to DDS_BOOLEAN_FALSE. Suppose you have a system in which the data packets being sent is very small. However, you want the data to be sent reliably, and the latency between the time that data is sent to the time that data is received is not an issue. However, the total network bandwidth between the DataWriter and DataReader applications is limited.

If the DataWriter sends a burst of data at a high rate, it is possible that it will overwhelm the limited bandwidth of the network. If you allocate enough space for the DataWriter to store the data burst being sent (see 7.5.22 RESOURCE_LIMITS QosPolicy on page 451), then you can use the push_on_write parameter of the DATA_WRITER_PROTOCOL QosPolicy to delay sending the data until the reliable DataReader asks for it.

By setting push_on_write to DDS_BOOLEAN_FALSE, when write() is called on the DataWriter, no data is actually sent. Instead data is stored in the DataWriter’s send queue. Periodically, Connext DDS will be sending heartbeats informing the DataReader about the data that is available. So every heartbeat period, the DataReader will realize that the DataWriter has new data, and it will send an ACK/NACK, asking for them.

When DataWriter receives the ACK/NACK packet, it will put together a package of data, up to the size set by the parameter max_bytes_per_nack_response, to be sent to the DataReader. This method not only self-throttles the send rate, but also uses network bandwidth more efficiently by eliminating redundant packet headers when combining several small packets into one larger one. Please note that the DataWriter will always send at least one sample.

7.5.5.9 Properties

This QosPolicy cannot be modified after the DataWriter is created.

Since it is only for DataWriters, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.
When setting the fields in this policy, the following rules apply. If any of these are false, Connext DDS returns DDS_RETCODE_INCONSISTENT_POLICY:

- $\text{min}_\text{nack}_\text{response}_\text{delay} \leq \text{max}_\text{nack}_\text{response}_\text{delay}$
- $\text{fast}_\text{heartbeat}_\text{period} \leq \text{heartbeat}_\text{period}$
- $\text{late}_\text{joiner}_\text{heartbeat}_\text{period} \leq \text{heartbeat}_\text{period}$
- $\text{low}_\text{watermark} < \text{high}_\text{watermark}$

If batching is disabled:
- $\text{heartbeats}_\text{per}_\text{max}_\text{samples} \leq \text{writer}_\text{qos}_\text{resource}_\text{limits}_\text{max}_\text{samples}$

If batching is enabled:
- $\text{heartbeats}_\text{per}_\text{max}_\text{samples} \leq \text{writer}_\text{qos}_\text{resource}_\text{limits}_\text{max}_\text{batches}$

### 7.5.5.10 Related QosPolicies

- 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570
- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.21 RELIABILITY QosPolicy on page 447

### 7.5.5.11 Applicable DDS Entities

- 7.3 DataWriters on page 287

### 7.5.5.12 System Resource Considerations

A high $\text{max}_\text{bytes}_\text{per}_\text{nack}_\text{response}$ may increase the instantaneous network bandwidth required to send a single burst of traffic for resending dropped packets.

### 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension)

This QosPolicy defines various settings that configure how DataWriters allocate and use physical memory for internal resources.

It includes the members in Table 7.46 DDS_DataWriterResourceLimitsQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.
### Table 7.46 DDS_DataWriterResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>initial_concurrent_blocking_threads</td>
<td>Initial number of threads that are allowed to concurrently block on the <code>write()</code> call on the same <code>DataWriter</code>.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_concurrent_blocking_threads</td>
<td>Maximum number of threads that are allowed to concurrently block on <code>write()</code> call on the same <code>DataWriter</code>.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_remote_reader_filters</td>
<td>Maximum number of remote <code>DataReaders</code> for which this <code>DataWriter</code> will perform content-based filtering.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_batches</td>
<td>Initial number of batches that a <code>DataWriter</code> will manage if batching is enabled.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_batches</td>
<td>Maximum number of batches that a <code>DataWriter</code> will manage if batching is enabled. When batching is enabled, the maximum number of DDS samples that a <code>DataWriter</code> can store is limited by this value and <code>max_samples</code> in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.</td>
</tr>
<tr>
<td>DDS_DataWriterResourceLimitsInstanceReplacementKind</td>
<td>instance_replacement</td>
<td>Sets the kinds of instances allowed to be replaced when a <code>DataWriter</code> reaches <code>max_instances</code> in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451. (See 7.5.6.1 Configuring DataWriter Instance Replacement on page 404.)</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>replace_empty_instances</td>
<td>Whether to replace empty instances during instance replacement. (See 7.5.6.1 Configuring DataWriter Instance Replacement on page 404.)</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>autoregister_instances</td>
<td>Whether to register automatically instances written with non-NIL handle that are not yet registered, which will otherwise return an error. This can be especially useful if the instance has been replaced.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_virtual_writers</td>
<td>Initial number of virtual writers supported by a <code>DataWriter</code>.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_virtual_writers</td>
<td>Maximum number of virtual writers supported by a <code>DataWriter</code>. Sets the maximum number of unique virtual writers supported by a <code>DataWriter</code>, where virtual writers are added when DDS samples are written with the virtual writer GUID. This field is especially relevant in the configuration of Persistence Service <code>DataWriters</code>, since they publish information on behalf of multiple virtual writers.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_remote_readers</td>
<td>The maximum number of remote readers supported by a <code>DataWriter</code>.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_app_ack_remote_readers</td>
<td>The maximum number of application-level acknowledging remote readers supported by a <code>DataWriter</code>.</td>
</tr>
</tbody>
</table>
Table 7.46 DDS_DataWriterResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>initial_active_topic_queries</td>
<td>Initial number of active topic queries a DataWriter will manage.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_active_topic_queries</td>
<td>Maximum number of active topic queries a DataWriter will manage. When topic queries are enabled, the maximum number of topic queries that a DataWriter can process at the same time is limited by this value.</td>
</tr>
<tr>
<td>DDS_AllocationSettings_t</td>
<td>writer_loaned_sample_allocation</td>
<td>Sets the allocation settings of the DataWriter-managed sample pool, when using Zero Copy transfer over shared memory or FlatData language binding. The number of samples loaned by a DataWriter via FooDataWriter's get_loan() operation is limited by the max_count in writer_loaned_sample_allocation. See Chapter 23 Sending Large Data on page 943.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>initialize_writer_loaned_sample</td>
<td>Determines whether or not to initialize members to default values in loaned samples returned by FooDataWriter's get_loan() operation, when using Zero Copy transfer over shared memory or FlatData language binding. See Chapter 23 Sending Large Data on page 943.</td>
</tr>
</tbody>
</table>

DataWriters must allocate internal structures to handle the simultaneous blocking of threads trying to call write() on the same DataWriter, for the storage used to batch small DDS samples, and for content-based filters specified by DataReaders.

Most of these internal structures start at an initial size and by default, will grow as needed by dynamically allocating additional memory. You may set fixed, maximum sizes for these internal structures if you want to bound the amount of memory that a DataWriter can use. By setting the initial size to the maximum size, you will prevent Connext DDS from dynamically allocating any memory after the creation of the DataWriter.

When setting the fields in this policy, the following rule applies. If this is false, Connext DDS returns DDS_RETCODE_INCONSISTENT_POLICY:

- **max_concurrent_blocking_threads >= initial_concurrent_blocking_threads**

The initial_concurrent_blocking_threads is used to allocate necessary initial system resources. If necessary, it will be increased automatically up to the max_concurrent_blocking_threads limit.

Every user thread calling write() on a DataWriter may use a semaphore that will block the thread when the DataWriter’s send queue is full. Because user code may set a timeout, each thread must use a different semaphore. See the max_blocking_time parameter of the 7.5.21 RELIABILITY QosPolicy on page 447. This QoS is offered so that the user application can control the dynamic allocation of system resources by Connext DDS.

If you do not mind if Connext DDS dynamically allocates semaphores when needed, then you can set the max_concurrent_blocking_threads parameter to some large value like MAX_INT. However, if you
know exactly how many threads will be calling \texttt{write()} on the same \textit{DataWriter}, and you do not want Con-
next DDS to allocate any system resources or memory after initialization, then you should set:

\begin{equation}
\text{max\_concurrent\_blocking\_threads} = \text{initial\_concurrent\_blocking\_threads} = \text{NUM}
\end{equation}

where \textit{NUM} is the number of threads that could possibly block concurrently.

Each \textit{DataWriter} can perform content-based data filtering for up to \texttt{max\_remote\_reader\_filters} number of \textit{DataReader}s.

Values for \texttt{max\_remote\_reader\_filters} may be.

- \texttt{0}: The \textit{DataWriter} will not perform filtering for any \textit{DataReader}, which means the \textit{DataReader} will have to filter the data itself.
- \texttt{1 to (2^{31}-2)}: The \textit{DataWriter} will filter for up to the specified number of \textit{DataReader}s. In addition, the \textit{DataWriter} will store the result of the filtering per DDS sample per \textit{DataReader}.
- \texttt{DDS\_LENGTH\_UNLIMITED} (default): The \textit{DataWriter} will filter for up to (2^{31}-2) \textit{DataReader}s. However, in this case, the \textit{DataWriter} will not store the filtering result per DDS sample per \textit{DataReader}. Thus, if a DDS sample is resent (such as due to a loss of reliable communication), the DDS sample will be filtered again.

For more information, see 5.4 ContentFilteredTopics on page 227.

\textbf{7.5.6.1 Configuring DataWriter Instance Replacement}

When the \texttt{max\_instances} limit (in the 7.5.6 DATA\_WRITER\_RESOURCE\_LIMITS QosPolicy (DDS Extension) on page 401) is reached, a \textit{DataWriter} will try to make space for a new instance by replacing an existing instance according to the instance replacement kind set in \texttt{instance\_replacement} in the 7.5.6 DATA\_WRITER\_RESOURCE\_LIMITS QosPolicy (DDS Extension) on page 401. For the sake of instance replacement, an instance is considered to be unregistered, disposed, or alive. The oldest instance of the specified kind, if such an instance exists, would be replaced with the new instance. Also, all DDS samples of a replaced instance must already have been acknowledged, such that removing the instance would not deprive any existing reader from receiving them.

Since an unregistered instance is one that a \textit{DataWriter} will not update any further, unregistered instances are replaced before any other instance kinds. This applies for all \texttt{instance\_replacement} kinds; for example, the ALIVE\_THEN\_DISPOSED kind would first replace unregistered, then alive, and then disposed instances. The rest of the kinds specify one or two kinds (e.g DISPOSED and ALIVE\_OR\_ DISPOSED). For the single kind, if no unregistered instances are replaceable, and no instances of the specified kind are replaceable, then the instance replacement will fail. For the others specifying multiple kinds, it either specifies to look for one kind first and then another kind (e.g. ALIVE\_THEN\_DISPOSED), meaning if the first kind is found then that instance will be replaced, or it will replace either of the kinds specified (e.g. ALIVE\_OR\_DISPOSED), whichever is older as determined by the time of instance regis-
tering, writing, or disposing.
If an acknowledged instance of the specified kind is found, the DataWriter will reclaim its resources for the new instance. It will also invoke the DataWriterListener’s on_instance_replaced() callback (if installed) and notify the user with the handle of the replaced instance, which can then be used to retrieve the instance key from within the callback. If no replaceable instances are found, the new instance will fail to be registered; the DataWriter may block, if the instance registration was done in the context of a write, or it may return with an out-of-resources return code.

In addition, replace_empty_instances (in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401) configures whether instances with no DDS samples are eligible to be replaced. If this is set, then a DataWriter will first try to replace empty instances, even before replacing unregistered instances.

7.5.6.2 Example

If there are multiple threads that can write on the same DataWriter, and the write() operation may block (based on reliability_qos.max_blocking_time and HISTORY settings), you may want to set initial_concurrent_blocking_threads to the most likely number of threads that will block on the same DataWriter at the same time, and set max_concurrent_blocking_threads to the maximum number of threads that could potentially block in the worst case.

7.5.6.3 Properties

This QosPolicy cannot be modified after the DataWriter is created.

Since it is only for DataWriters, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

7.5.6.4 Related QosPolicies

- 7.5.2 BATCH QosPolicy (DDS Extension) on page 374
- 7.5.21 RELIABILITY QosPolicy on page 447
- 7.5.12 HISTORY QosPolicy on page 420

7.5.6.5 Applicable DDS Entities

- 7.3 DataWriters on page 287

7.5.6.6 System Resource Considerations

Increasing the values in this QosPolicy will cause more memory usage and more system resource usage.

7.5.7 DEADLINE QosPolicy

On a DataWriter, this QosPolicy states the maximum period in which the application expects to call write() on the DataWriter, thus publishing a new DDS sample. The application may call write() faster than the
rate set by this QosPolicy.

On a DataReader, this QosPolicy states the maximum period in which the application expects to receive new values for the Topic. The application may receive data faster than the rate set by this QosPolicy.

The DEADLINE QosPolicy has a single member, shown in Table 7.47 DDS_DeadlineQosPolicy. For the default and valid range, please refer to the API Reference HTML documentation.

Table 7.47 DDS_DeadlineQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>period</td>
<td>For DataWriters: maximum time between writing a new value of an instance.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For DataReaders: maximum time between receiving new values for an instance.</td>
</tr>
</tbody>
</table>

You can use this QosPolicy during system integration to ensure that applications have been coded to meet design specifications. You can also use it during run time to detect when systems are performing outside of design specifications. Receiving applications can take appropriate actions to prevent total system failure when data is not received in time. For topics on which data is not expected to be periodic, the deadline period should be set to an infinite value.

For keyed topics, the DEADLINE QoS applies on a per-instance basis. An application must call write() for each known instance of the Topic within the period specified by the DEADLINE on the DataWriter or receive a new value for each known instance within the period specified by the DEADLINE on the DataReader. For a DataWriter, the deadline period begins when the instance is first written or registered. For a DataReader, the deadline period begins when the first DDS sample is received.

Connext DDS will modify the OFFERED_DEADLINE_MISSED_STATUS and call the associated method in the DataWriterListener (see 7.3.6.5 OFFERED_DEADLINE_MISSED Status on page 303) if the application fails to write() a value for an instance within the period set by the DEADLINE QosPolicy of the DataWriter.

Similarly, Connext DDS will modify the REQUESTED_DEADLINE_MISSED_STATUS and call the associated method in the DataReaderListener (see 8.3.7.5 REQUESTED_DEADLINE_MISSED Status on page 526) if the application fails to receive a value for an instance within the period set by the DEADLINE QosPolicy of the DataReader.

For DataReaders, the DEADLINE QosPolicy and the 8.6.4 TIME_BASED_FILTER QosPolicy on page 588 may interact such that even though the DataWriter writes DDS samples fast enough to fulfill its commitment to its own DEADLINE QosPolicy, the DataReader may see violations of its DEADLINE QosPolicy. This happens because Connext DDS will drop any packets received within the minimum separation set by the TIME_BASED_FILTER—packets that could satisfy the DataReader’s deadline.

To avoid triggering the DataReader’s deadline even though the matched DataWriter is meeting its own deadline, set your QoS parameters to meet the following relationship:
Although you can set the DEADLINE QosPolicy on Topics, its value can only be used to initialize the DEADLINE QosPolicies of either a DataWriter or DataReader. It does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.

7.5.7.1 Example

Suppose you have a time-critical piece of data that should be updated at least once every second. You can set the DEADLINE period to 1 second on both the DataWriter and DataReader. If there is no update within that time, the DataWriter will get an on_offered_deadline_missed Listener callback, and the DataReader will get on_requested_deadline_missed, so that both sides can handle the error situation properly.

Note that in practice, there will be latency and jitter in the time between when data is send and when data is received. Thus even if the DataWriter is sending data at exactly 1 second intervals, the DataReader may not receive the data at exactly 1 second intervals. More likely, it will DataReader will receive the data at 1 second plus a small variable quantity of time. Thus you should accommodate this practical reality in choosing the DEADLINE period as well as the actual update period of the DataWriter or your application may receive false indications of failure.

The DEADLINE QosPolicy also interacts with the OWNERSHIP QosPolicy when OWNERSHIP is set to EXCLUSIVE. If a DataReader fails to receive data from the highest strength DataWriter within its requested DEADLINE, then the DataReaders can fail-over to lower strength DataWriters, see the 7.5.17 OWNERSHIP QosPolicy on page 434.

7.5.7.2 Properties

This QosPolicy can be changed at any time.

The deadlines on the two sides must be compatible.

*DataWriter’s* DEADLINE period <= *DataReader’s* DEADLINE period.

That is, the DataReader cannot expect to receive DDS samples more often than the DataWriter commits to sending them.

If the DataReader and DataWriter have compatible deadlines, Connext DDS monitors this “contract” and informs the application of any violations. If the deadlines are incompatible, both sides are informed and communication does not occur. The ON_OFFERED_INCOMPATIBLE_QOS and the ON_REQUESTED_INCOMPATIBLE_QOS statuses will be modified and the corresponding Listeners called for the DataWriter and DataReader respectively.
7.5.7.3 Related QoS Policies

- 7.5.15 LIVELINESS QoSPolicy on page 427
- 7.5.17 OWNERSHIP QoSPolicy on page 434
- 8.6.4 TIME_BASED_FILTER QoSPolicy on page 588

7.5.7.4 Applicable DDS Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.7.5 System Resource Considerations

A Connext DDS-internal thread will wake up at least by the DEADLINE period to check to see if the deadline was missed. It may wake up faster if the last DDS sample that was published or sent was close to the last time that the deadline was checked. Therefore a short period will use more CPU to wake and execute the thread checking the deadline.

7.5.8 DESTINATION_ORDER QoSPolicy

When multiple DataWriters send data for the same topic, the order in which data from different DataWriters are received by the applications of different DataReaders may be different. Thus different DataReaders may not receive the same "last" value when DataWriters stop sending data.

This policy controls how each subscriber resolves the final value of a data instance that is written by multiple DataWriters (which may be associated with different Publishers) running on different nodes.

This QoSPolicy can be used to create systems that have the property of "eventual consistency." Thus intermediate states across multiple applications may be inconsistent, but when DataWriters stop sending changes to the same topic, all applications will end up having the same state.

Each DDS sample includes two timestamps: a source timestamp and a reception timestamp. The source timestamp is recorded by the DataWriter application when the data was written. The reception timestamp is recorded by the DataReader application when the data was received.

This QoS includes the members in Table 7.48 DDS_DestinationOrderQoSPolicy.
This QoS policy can be set for both DataWriters and DataReaders. See 7.5.8.1 Properties on the next page for compatibility rules.

For a DataReader:

- DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS

  Assuming the OWNERSHIP_STRENGTH allows it, the latest received value for the instance should be the one whose value is kept. Data will be delivered by a DataReader in the order in which it was received (which may lead to inconsistent final values).

- DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS

  If scope is set to DDS_INSTANCE_SCOPE_DESTINATIONORDER_QOS (default), within each instance, the sample's source timestamp shall be used to determine the most recent information. This is the only setting that, in the case of concurrent same-strength DataWriters updating the same instance, ensures that all DataReaders end up with the same final value for the instance. If a DataReader receives a sample for an instance with a source timestamp that is older than the last source timestamp received for the instance, the sample is dropped. You can keep track of the total

| DDS_DestinationOrderQosPolicyKind | kind | Can be either:
| DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS  
DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS |
| DDS_DestinationOrderQosPolicyScopeKind | scope | Can be either:
| DDSINSTANCE_SCOPE_DESTINATIONORDER_QOS - Indicates that data is ordered on a per instance basis if used along with DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS. The source timestamp of the current sample is compared to the source timestamp of the previously received sample for the same instance. The tolerance check is also applied per instance.
DDS_TOPIC_SCOPE_DESTINATIONORDER_QOS - Indicates that data is ordered on a per topic basis if used along with DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS. The source timestamp of the current sample is compared to the source timestamp of the previously received sample for the same topic. The tolerance check is also applied per topic. |
| DDS_Duration_t | source_timestamp_tolerance | Allowed tolerance between source timestamps of consecutive DDS samples. Only applies when kind (above) is DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS.

  - For a DataWriter: The write operation will fail if the source timestamp of the sample is older than the timestamp of the previously written DDS sample by more than the source_timestamp_tolerance.
  - For a DataReader: A DataReader will accept a sample only if the source timestamp is no farther in the future from the reception timestamp than the source Timestamp_tolerance.
number of dropped samples for this reason with the old_source_timestamp_dropped_sample_count field in the 8.3.7.2 DATA_READER_CACHE_STATUS on page 517. The SAMPLE_REJECTED status or the SAMPLE_LOST status will not be updated.

If scope is set to DDS_TOPIC_SCOPE_DESTINATIONORDER_QOS, the ordering is enforced per topic across all instances.

In addition, a DataReader will accept a sample only if the source timestamp is no farther in the future from the reception timestamp than the source_timestamp_tolerance. Otherwise, the DDS sample is dropped. You can keep track of the total number of dropped samples for this reason with the tolerance_source_timestamp_dropped_sample_count field in the 8.3.7.2 DATA_READER_CACHE_STATUS on page 517. The SAMPLE_REJECTED status or the SAMPLE_LOST status will not be updated.

For the DataWriter:

- DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS

  The DataWriter will not enforce source timestamp ordering when writing samples using the DataWriter::write_w_timestamp or DataWriter::write_w_params API. The source timestamp of a new sample can be older than the source timestamp of the previous samples.

- DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS

  If scope is set to DDS_INSTANCE_SCOPE_DESTINATIONORDER_QOS (default), when writing a sample, the sample’s timestamp must not be older than the timestamp of the previously written DDS sample for the same instance. If, however, the timestamp is older than the timestamp of the previously written DDS sample—but the difference is less than the source_timestamp_tolerance—the DDS sample will use the previously written DDS sample's timestamp as its timestamp. Otherwise, if the difference is greater than the tolerance, the write will fail with retcode DDS_RETCODE_BAD_PARAMETER.

  If scope is set to DDS_TOPIC_SCOPE_DESTINATIONORDER_QOS, a new sample timestamp must not be older than the timestamp of the previously written DDS sample, across all instances. (The ordering is enforced across all instances.)

Although you can set the DESTINATION_ORDER QosPolicy on Topics, its value can only be used to initialize the DESTINATION_ORDER QosPolicies of either a DataWriter or DataReader. It does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.

7.5.8.1 Properties

This QosPolicy cannot be modified after the Entity is enabled.
This QoS must be set compatibly between the DataWriter and the DataReader. The compatible combinations are shown in Table 7.49 Valid Reader/Writer Combinations of DestinationOrder.

### Table 7.49 Valid Reader/Writer Combinations of DestinationOrder

<table>
<thead>
<tr>
<th>Destination Order</th>
<th>DataReader requests:</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BY_SOURCE</td>
<td>compatible</td>
<td>compatible</td>
</tr>
<tr>
<td>DataWriter offers:</td>
<td>BY_SOURCE</td>
<td>compatible</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BY_RECEPTION</td>
<td>incompatible</td>
<td>compatible</td>
</tr>
</tbody>
</table>

If this QosPolicy is set incompatibly, the ON_OFFERED_INCOMPATIBLE_QOS and ON_REQUESTED_INCOMPATIBLE_QOS statuses will be modified and the corresponding Listeners called for the DataWriter and DataReader respectively.

#### 7.5.8.2 Related QosPolicies

- 7.5.17 OWNERSHIP QosPolicy on page 434
- 7.5.12 HISTORY QosPolicy on page 420

#### 7.5.8.3 Applicable DDS Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

#### 7.5.8.4 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

#### 7.5.9 DURABILITY QosPolicy

Because the publish-subscribe paradigm is connectionless, applications can create publications and subscriptions in any way they choose. As soon as a matching pair of DataWriters and DataReaders exist, the data published by the DataWriter will be delivered to the DataReader. However, a DataWriter may publish data before a DataReader has been created. For example, before you subscribe to a magazine, there have been past issues that were published.

The DURABILITY QosPolicy controls whether or not, and how, published DDS samples are stored by the DataWriter application for DataReaders that are found after the DDS samples were initially written. DataReaders use this QoS to request DDS samples that were published before they were created. The analogy is for a new subscriber to a magazine to ask for issues that were published in the past. These are
known as ‘historical’ DDS data samples. (Reliable DataReaders may wait for these historical DDS samples, see 8.3.5 Checking DataReader Status and StatusConditions on page 514.)

This QosPolicy can be used to help ensure that DataReaders get all data that was sent by DataWriters, regardless of when it was sent. This QosPolicy can increase system tolerance to failure conditions.

The 7.5.12 HISTORY QosPolicy on page 420 controls how many samples the DataWriter stores for repair to currently matched DataReaders. The DURABILITY QosPolicy controls how many samples the DataWriter stores for sending to late-joining DataReaders (DataReaders that are found after the samples were initially written). See Figure 7.30: History Depth and Durability Depth on page 424.

See also Mechanisms for Achieving Information Durability and Persistence (Chapter 13 on page 732).

The possible settings for this QoS are:

- **DDS_VOLATILE_DURABILITY_QOS**
  
  Connext DDS is not required to send and will not deliver any DDS data samples to DataReaders that are discovered after the DDS samples were initially published.

- **DDS_TRANSIENT_LOCAL_DURABILITY_QOS**
  
  Connext DDS will store and send previously published DDS samples for delivery to newly discovered DataReaders as long as the DataWriter still exists. For this setting to be effective, you must also set the 7.5.21 RELIABILITY QosPolicy on page 447 kind to Reliable (not Best Effort). Which particular DDS samples are kept depends on other QoS settings such as 7.5.12 HISTORY QosPolicy on page 420 and 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.

- **DDS_TRANSIENT_DURABILITY_QOS**
  
  Connext DDS will store previously published DDS samples in memory using Persistence Service, which will send the stored data to newly discovered DataReaders. Which particular DDS samples are kept and sent by Persistence Service depends on the 7.5.12 HISTORY QosPolicy on page 420 and 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 of the Persistence Service DataWriters. These QosPolicies can be configured in the Persistence Service configuration file or through the 7.5.10 DURABILITY SERVICE QosPolicy on page 416 of the DataWriters configured with DDS_TRANSIENT_DURABILITY_QOS.

- **DDS_PERSISTENT_DURABILITY_QOS**
  
  Connext DDS will store previously published DDS samples in permanent storage, like a disk, using Persistence Service, which will send the stored data to newly discovered DataReaders. Which particular DDS samples are kept and sent by Persistence Service depends on the 7.5.12 HISTORY QosPolicy on page 420 and 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 in the Persistence Service DataWriters. These QosPolicies can be configured in the Persistence Service
This QosPolicy includes the members in Table 7.50 DDS_DurabilityQosPolicy. For default settings, please refer to the API Reference HTML documentation.

Table 7.50 DDS_DurabilityQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| DDS_DurabilityQosPolicyKind | kind                  | DDS_VOLATILE_DURABILITY_QOS: Do not save or deliver historical DDS samples.  
-DDS_TRANSIENT_LOCAL_DURABILITY_QOS: Save and deliver historical DDS samples if the DataWriter still exists.  
-DDS_TRANSIENT_DURABILITY_QOS: Save and deliver historical DDS samples using Persistence Service to store samples in volatile memory.  
-DDS_PERSISTENCE_DURABILITY_QOS: Save and deliver historical DDS samples using Persistence Service to store samples in non-volatile memory. |
| DDS_Long     | writer_depth          | How many DDS samples are stored per instance by the DataWriter application for sending to late-joining DataReaders (DataReaders that are found after the DDS samples were initially written).  
The default value, AUTO, makes this parameter equal to the following:  
- History depth in the 7.5.12 HISTORY QosPolicy on page 420 if the History kind is KEEP_LAST.  
- max_samples_per_instance in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 if the History kind is KEEP_ALL.  
The writer_depth must be <= to the History depth in the HISTORY QosPolicy if the History kind is KEEP_LAST.  
writer_depth applies only to non-volatile DataWriters (those for which the kind is DDS_TRANSIENT_LOCAL_DURABILITY_QOS, DDS_TRANSIENT_DURABILITY_QOS, or DDS_PERSISTENCE_DURABILITY_QOS).  
writer_depth set on the DataReader side will be ignored. |
| DDS_Boolean  | direct_communication   | Whether or not a TRANSIENT or PERSISTENT DataReader should receive DDS samples directly from a TRANSIENT or PERSISTENT DataWriter.  
When TRUE, a TRANSIENT or PERSISTENT DataReader will receive DDS samples directly from the original DataWriter. The DataReader may also receive DDS samples from Persistence Service, but the duplicates will be filtered by the middleware.  
When FALSE, a TRANSIENT or PERSISTENT DataReader will receive DDS samples only from the DataWriter created by Persistence Service. This 'relay communication' pattern provides a way to guarantee eventual consistency.  
See 13.5.1 RTI Persistence Service on page 749.  
This field only applies to DataReaders. |
With this QoS policy alone, there is no way to specify or characterize the intended consumers of the information. With TRANSIENT_LOCAL, TRANSIENT, or PERSISTENT durability a DataWriter can be configured to keep DDS samples around for late-joiners. However, there is no way to know when the information has been consumed by all the intended recipients.

Information durability can be combined with required subscriptions in order to guarantee that DDS samples are delivered to a set of required subscriptions. For additional details on required subscriptions see 7.3.13 Required Subscriptions on page 323 and 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370.

A DataWriter will keep at most History.depth samples per instance until they are fully acknowledged. Samples outside of the Durability.writer_depth for an instance will be removed once they are fully acknowledged. Only the most recent Durability.writer_depth samples per instance will be kept by the DataWriter for delivery to late-joining non-volatile DataReaders.

When writer_depth is used in combination with batching, it acts as a minimum number of samples that will be kept per instance, rather than a maximum. Any batch that contains a sample that falls within the writer_depth of the instance to which it belongs will be sent to late-joining DataReaders. This means that batches may be sent that contain samples from other instances, or the same instance, that fall outside of the writer_depth for the instance to which they belong. For example, if the writer_depth is set to 1 and a batch with two samples for the same instance is written, then when a late-joining DataReader is discovered, the DataWriter will send the batch containing two samples for the same instance to the DataReader.

7.5.9.1 Example

Suppose you have a DataWriter that sends data sporadically and its DURABILITY kind is set to VOLATILE. If a new DataReader joins the system, it won’t see any data until the next time that write() is called on the DataWriter. If you want the DataReader to receive any data that is valid, old or new, both sides should set their DURABILITY kind to TRANSIENT_LOCAL. This will ensure that the DataReader gets some of the previous DDS samples immediately after it is enabled.

7.5.9.2 Properties

This QosPolicy cannot be modified after the Entity has been created.

The DataWriter and DataReader must use compatible settings for this QosPolicy. To be compatible, the DataWriter and DataReader must use one of the valid combinations shown in Table 7.51 Valid Combinations of Durability ‘kind’.

If this QosPolicy is found to be incompatible, the ON_OFFERED_INCOMPATIBLE_QOS and ON_REQUESTED_INCOMPATIBLE_QOS statuses will be modified and the corresponding Listeners called for the DataWriter and DataReader respectively.
Table 7.51 Valid Combinations of Durability ‘kind’

<table>
<thead>
<tr>
<th>DataWriter offers:</th>
<th>VOLATILE</th>
<th>TRANSIENT_LOCAL</th>
<th>TRANSIENT</th>
<th>PERSISTENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOLATILE</td>
<td>compatible</td>
<td>incompatible</td>
<td>incompatible</td>
<td>incompatible</td>
</tr>
<tr>
<td>TRANSIENT_LOCAL</td>
<td>compatible</td>
<td>compatible</td>
<td>incompatible</td>
<td>incompatible</td>
</tr>
<tr>
<td>TRANSIENT</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
<td>incompatible</td>
</tr>
<tr>
<td>PERSISTENT</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
</tr>
</tbody>
</table>

7.5.9.3 Related QosPolicies

- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.21 RELIABILITY QosPolicy on page 447
- 7.5.10 DURABILITY SERVICE QosPolicy on the next page
- 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370

7.5.9.4 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.9.5 System Resource Considerations

Using this policy with a setting other than VOLATILE will cause Connext DDS to use CPU and network bandwidth to send old DDS samples to matching, newly discovered DataReaders. The actual amount of resources depends on the total size of data that needs to be sent.

The maximum number of DDS samples that will be kept on the DataWriter’s queue for late-joiners and/or required subscriptions is determined by max_samples in RESOURCE_LIMITS Qos Policy.

System Resource Considerations With Required Subscriptions

By default, when TRANSIENT_LOCAL durability is used in combination with required subscriptions, a DataWriter configured with KEEP_ALL in the 7.5.12 HISTORY QosPolicy on page 420 will keep the DDS samples in its cache until they are acknowledged by all the required subscriptions. (For additional details, see 7.3.13 Required Subscriptions on page 323.) After the DDS samples are acknowledged by the required subscriptions they will be marked as reclaimable, but they will not be purged from the
DataWriter’s queue until the DataWriter needs these resources for new DDS samples. This may lead to a non efficient resource utilization, specially when max_samples is high or even UNLIMITED.

The DataWriter’s behavior can be changed to purge DDS samples after they have been acknowledged by all the active/matching DataReaders and all the required subscriptions configured on the DataWriter. To do so, set the dds.data_writer.history.purge_samples_after_acknowledgment property to 1 (see 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439).

See 7.3.13 Required Subscriptions on page 323.

7.5.10 DURABILITY SERVICE QosPolicy

This QosPolicy is only used if the 7.5.9 DURABILITY QosPolicy on page 411 is PERSISTENT or TRANSIENT and you are using Persistence Service. It is used to store and possibly forward the data sent by the DataWriter to DataReaders that are created after the data was initially sent.

This QosPolicy configures certain parameters of Persistence Service when it operates on the behalf of the DataWriter, such as how much data to store. Specifically, this QosPolicy configures the HISTORY and RESOURCE_LIMITS used by the fictitious DataReader and DataWriter used by Persistence Service.

Note however, that by default, Persistence Service will ignore the values in the 7.5.10 DURABILITY SERVICE QosPolicy above and must be configured to use those values.

For more information, please see:

- Mechanisms for Achieving Information Durability and Persistence (Chapter 13 on page 732)
- Introduction to RTI Persistence Service (Chapter 45 on page 1180)
- Configuring Persistence Service (Chapter 46 on page 1181)

This QosPolicy includes the members in Table 7.52 DDS_DurabilityServiceQosPolicy. For default values, please refer to the API Reference HTML documentation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>service_cleanup_delay</td>
<td>How long to keep all information regarding an instance. Can be: Zero (default): Purge disposed instances from Persistence Service immediately. However, this will only happen if use_durability_service = 1. INFINITE: Do not purge disposed instances.</td>
</tr>
<tr>
<td>DDS_HistoryQosPolicyKind</td>
<td>history_kind</td>
<td>Setting to use for the 7.5.12 HISTORY QosPolicy on page 420 kind when recouping durable data.</td>
</tr>
</tbody>
</table>
Table 7.52 DDS_DurabilityServiceQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>history_depth</td>
<td>Setting to use for the 7.5.9 DURABILITY QosPolicy on page 411 writer_depth when recouping durable data. If the 7.5.12 HISTORY QosPolicy on page 420 depth is set to a value lower than this value, then the HISTORY depth will be set equal to the value of this field.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_samples</td>
<td>Settings to use for the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 when feeding data to a late joiner.</td>
</tr>
<tr>
<td></td>
<td>max_instances</td>
<td></td>
</tr>
<tr>
<td></td>
<td>max_samples_per_instance</td>
<td></td>
</tr>
</tbody>
</table>

The service_cleanup_delay in this QosPolicy controls when Persistence Service may remove all information regarding a data-instance. Information on a data-instance is maintained until all of the following conditions are met:

1. The instance has been explicitly disposed
   (instance_state = NOT_ALIVE_DISPOSED).
2. All samples for the disposed instance have been acknowledged, including the dispose sample itself.
3. A time interval longer that DurabilityService QosPolicy’s service_cleanup_delay has elapsed since the time that Connext DDS detected that the previous two conditions were met. (Note: Only values of zero or INFINITE are currently supported for service_cleanup_delay.)

The service_cleanup_delay field is useful in the situation where your application disposes an instance and it crashes before it has a chance to complete additional tasks related to the disposition. Upon restart, your application may ask for initial data to regain its state and the delay introduced by service_cleanup_delay will allow your restarted application to receive the information about the disposed instance and complete any interrupted tasks.

Although you can set the DURABILITY_SERVICE QosPolicy on a Topic, this is only useful as a means to initialize the DURABILITY_SERVICE QosPolicy of a DataWriter. A Topic’s DURABILITY_SERVICE setting does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.

7.5.10.1 Properties

This QosPolicy cannot be modified after the Entity has been enabled.
It does not apply to *DataReaders*, so there is no requirement for setting it compatibly on the sending and receiving sides.

### 7.5.10.2 Related QosPolicies

- 7.5.9 DURABILITY QosPolicy on page 411
- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.22 RESOURCE_LIMITS QosPolicy on page 451

### 7.5.10.3 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287

### 7.5.10.4 System Resource Considerations

Since this QosPolicy configures the HISTORY and RESOURCE_LIMITS used by the fictitious *DataReader* and *DataWriter* used by *Persistence Service*, it does have some impact on resource usage.

## 7.5.11 ENTITY_NAME QosPolicy (DDS Extension)

The ENTITY_NAME QosPolicy assigns a name and role name to a *DomainParticipant*, *Publisher*, *Subscriber*, *DataReader*, or *DataWriter*.

How the **name** is used is strictly application-dependent.

It is useful to attach names that are meaningful to the user. These names (except for *Publishers* and *Subscribers*) are propagated during discovery so that applications can use these names to identify, in a user-context, the entities that it discovers. Also, *RTI Connext* tools will print the names of discovered entities (except for *Publishers* and *Subscribers*).

The **role_name** identifies the role of the entity. It is used by the Collaborative DataWriter feature (see 7.5.1.1 Availability QoS Policy and Collaborative DataWriters on page 371). With Durable Subscriptions, **role_name** is used to specify to which Durable Subscription the *DataReader* belongs. (see 7.5.1.2 Availability QoS Policy and Required Subscriptions on page 372).

This QosPolicy contains the members listed in Table 7.53 DDS_EntityNameQoSPolicy.
Table 7.53 DDS_EntityNameQoS Policy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>name</td>
<td>A null-terminated string up to 255 characters in length.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To set this in XML, see 19.4.8 Entity Names on page 882.</td>
</tr>
<tr>
<td>char</td>
<td>role_name</td>
<td>A null-terminated string up to 255 characters in length.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>To set this in XML, see 19.4.8 Entity Names on page 882.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For Collaborative DataWriters, this name is used to specify to which</td>
</tr>
<tr>
<td></td>
<td></td>
<td>endpoint group the DataWriter belongs. See 7.5.1.1 Availability QoS Policy</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and Collaborative DataWriters on page 371.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For Required and Durable Subscriptions this name is used to specify to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>which Subscription the DataReader belongs. See 7.3.13 Required Subscriptions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>on page 323.</td>
</tr>
</tbody>
</table>

These names will appear in the built-in topic for the entity (see the tables in 18.2 Built-in DataReaders on page 832).

Prior to get_qos(), if the name and/or role_name field in this QoS Policy is not null, Connext DDS assumes the memory to be valid and big enough and may write to it. If that is not desired, set name and/or role_name to NULL before calling get_qos() and Connext DDS will allocate adequate memory for name.

When you call the destructor of entity’s QoS structure (DomainParticipantQos, DataReaderQos, or DataWriterQos) (in C++, C++/CLI, and C#) or <entity>Qos_finalize() (in C), Connext DDS will attempt to free the memory used for name and role_name if it is not NULL. If this behavior is not desired, set name and/or role_name to NULL before you call the destructor of entity’s QoS structure or DomainParticipantQos_finalize().

7.5.11.1 Properties

This QoS Policy cannot be modified after the entity is enabled.

7.5.11.2 Related QoS Policies

- None

7.5.11.3 Applicable Entities

- 9.3 DomainParticipants on page 609
- 7.2 Publishers on page 271
- 8.2 Subscribers on page 486
- 8.3 DataReaders on page 504
- 7.3 Data Writers on page 287
7.5.11.4 System Resource Considerations

If the value of `name` in this QosPolicy is not NULL, some memory will be consumed in storing the information in the database, but should not significantly impact the use of resource.

7.5.12 HISTORY QosPolicy

This QosPolicy configures the number of DDS samples that Connext DDS will store locally for DataWriters and DataReaders. For reliable DataWriters, the HISTORY QosPolicy configures the reliability window, or the number of samples that are kept until all matching DataReaders have fully-acknowledged the samples. For keyed Topics, this QosPolicy applies on a per instance basis, so that Connext DDS will attempt to store the configured value of DDS samples for every instance (see 2.4 DDS Samples, Instances, and Keys on page 18 for a discussion of keys and instances).

This QoS policy includes the members seen in Table 7.54 DDS_HistoryQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

<table>
<thead>
<tr>
<th>Table 7.54 DDS_HistoryQosPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

The `kind` determines whether or not to save a configured number of DDS samples or all DDS samples. In either case, when using a Reliable 7.5.21 RELIABILITY QosPolicy on page 447, the samples are kept until they are fully acknowledged by all matching DataReaders. Once a sample is fully acknowledged, it is removed from the DataWriter's queue, unless it needs to be kept for durability purposes. (See 7.5.9 DURABILITY QosPolicy on page 411). The HISTORY QoS Policy `kind` can be set to either of the following:

- **DDS_KEEP_LAST_HISTORY_QOS.** Connext DDS attempts to keep the latest values of the data-instance and discard the oldest ones when the limit as set by the depth parameter is reached; new data will overwrite the oldest data in the queue. Thus the queue acts like a circular buffer of length `depth`. Invalid samples are samples representing the disposal or unregistration of an instance. There is only ever one invalid sample per-instance and that one sample can be in different states...
depending on whether the instance has been disposed, unregistered, or both. How invalid samples affect the history depth differs for DataReaders and DataWriters:

- For a DataWriter: Connext DDS attempts to keep the most recent depth DDS samples of each instance (identified by a unique key) managed by the DataWriter. Invalid samples count towards the depth and may replace other DDS samples currently in the DataWriter queue.

- For a DataReader: Connext DDS attempts to keep the most recent depth DDS samples received for each instance (identified by a unique key) until the application takes them via the DataReader's take() operation. See 8.4.3 Accessing DDS Data Samples with Read or Take on page 555 for a discussion of the difference between read() and take(). Invalid samples do not count towards the depth and will not replace other DDS samples currently in the DataReader queue.

- DDS_KEEP_ALL_HISTORY_QOS. Connext DDS attempts to keep all of the DDS samples of a Topic.

  - For a DataWriter: Connext DDS attempts to keep all DDS samples published by the DataWriter.

  - For a DataReader: Connext DDS attempts to keep all DDS samples received by the DataReader for a Topic (both keyed and non-keyed) until the application takes them via the DataReader's take() operation. See 8.4.3 Accessing DDS Data Samples with Read or Take on page 555 for a discussion of the difference between read() and take().

  - The value of the depth parameter is ignored.

The above descriptions say “attempts to keep” because the actual number of DDS samples kept is subject to the limitations imposed by the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451. All of the DDS samples of all instances of a Topic share a single physical queue that is allocated for a DataWriter or DataReader. The size of this queue is configured by the RESOURCE_LIMITS QosPolicy. If there are many different instances for a Topic, it is possible that the physical queue may run out of space before the number of DDS samples reaches the depth for all instances.

In the KEEP_ALL case, Connext DDS can only keep as many DDS samples for a Topic (independent of instances) as the size of the allocated queue. Connext DDS may or may not allocate more memory when the queue is filled, depending on the settings in the RESOURCE_LIMITS QoSPolicy of the DataWriter or DataReader.

This QosPolicy interacts with the 7.5.21 RELIABILITY QosPolicy on page 447 by controlling whether or not Connext DDS guarantees that ALL of the data sent is received or if only the last N data values sent are guaranteed to be received (a reduced level of reliability using the KEEP_LAST setting). However, the physical sizes of the send and receive queues are not controlled by the History QosPolicy. The memory allocation for the queues is controlled by the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.

What happens when the physical queue is filled depends both on the setting for the HISTORY QosPolicy as well as the RELIABILITY QosPolicy.
• **DDS_KEEP_LAST_HISTORY_QOS**
  
  • If RELIABILITY is **BEST EFFORT**: When the number of DDS samples for an instance in the queue reaches the value of *depth*, a new DDS sample for the instance will replace the oldest DDS sample for the instance in the queue.

  • If RELIABILITY is **RELIABLE**: When the number of DDS samples for an instance in the queue reaches the value of *depth*, a new DDS sample for the instance will replace the oldest DDS sample for the instance in the queue—even if the DDS sample being overwritten has not been fully acknowledged as being received by all reliable *DataReaders*. This implies that the discarded DDS sample may be lost (with the reason LOST_BY_WRITER) by some reliable *DataReaders*. Thus, when using the **KEEP_LAST** setting, strict reliability is not guaranteed. See **Reliable Communications** (Chapter 11 on page 688) for a complete discussion on Connext DDS’s reliable protocol.

• **DDS_KEEP_ALL_HISTORY_QOS**
  
  • If RELIABILITY is **BEST EFFORT**: For a *DataWriter*, if the number of DDS samples for an instance in the queue reaches the value of the *max_samples_per_instance* field, a new DDS sample for the instance will replace the oldest DDS sample for the instance in the queue (regardless of instance). For a *DataReader*, a new DDS sample received by the *DataReader* when this resource limit is exceeded will be lost with the reason DDS_LOST_BY_SAMPLES_PER_INSTANCE_LIMIT.

  • If RELIABILITY is **RELIABLE**: When the number of DDS samples for an instance in the queue reaches the value of the *max_samples_per_instance* field, then:
    
    • For a *DataWriter*: A new DDS sample for the instance will replace the oldest DDS sample for the instance in the sending queue—only if the DDS sample being overwritten has been fully acknowledged as being received by all reliable *DataReaders*. If the oldest DDS sample for the instance has not been fully acknowledged, the *write()* operation trying to enter a new DDS sample for the instance into the sending queue will block (for the *max_blocking_time* specified in the RELIABLE QosPolicy).

    • For a *DataReader*: *max_samples_per_instance* represents the maximum number of DDS samples of any one instance that are stored in the *DataReader* output queue—that is, the queue from which the application takes/reads samples. Therefore, when *max_samples_per_instance* is hit, the *DataWriter* samples will be rejected. They will not be moved to the *DataReader* output queue. They will stay in the *DataWriter's* remote writer queue until there is space for them in the *DataReader* output queue (until the samples in the *DataReader* output queue are taken). On a reliable *DataReader*, there is one remote writer queue for each *DataWriter* that matches the *DataReader*. The remote writer queue size is configurable with the resource limit **reader_qos.reader_resource_**
7.5.12 HISTORY QosPolicy

**limit.max_samples_per_remote_writer** (see 8.6.2 DATA_READER_ RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576).

Although you can set the HISTORY QosPolicy on Topics, its value can only be used to initialize the HISTORY QosPolicies of either a DataWriter or DataReader. It does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.

**7.5.12.1 Example**

To achieve strict reliability, you must (1) set the DataWriter’s and DataReader’s HISTORY QosPolicy to KEEP_ALL, and (2) set the DataWriter’s and DataReader’s RELIABILITY QosPolicy to RELIABLE.

See Reliable Communications (Chapter 11 on page 688) for a complete discussion on Connext DDS’s reliable protocol.

See 11.3.3 Controlling Queue Depth with the History QosPolicy on page 703.

It is possible to configure the reliability window (the number of samples kept in the queue for reliability purposes) separately from the durability window (the number of samples kept in the DataWriter queue for late-joining DataReaders). This allows an application to achieve the level of reliability that is required and still only deliver a subset of data to late-joining DataReaders when using a non-VOLATILE 7.5.9 DURABILITY QosPolicy on page 411. Figure 7.30: History Depth and Durability Depth on the next page shows the relationship between History depth and Durability writer_depth.
The History depth determines how many samples to keep for reliability purposes (for example, for redelivering to DataReaders that haven’t acknowledged them yet). The 7.5.9 DURABILITY QosPolicy on page 411 writer_depth determines what subset of those samples to deliver to late-joining DataReaders.

7.5.12.2 Properties

This QosPolicy cannot be modified after the Entity has been enabled.

There is no requirement that the publishing and subscribing sides use compatible values.

7.5.12.3 Related QosPolicies

- 7.5.2 BATCH QosPolicy (DDS Extension) on page 374 Do not configure the DataReader’s depth to be shallower than the DataWriter's maximum batch size (batch_max_data_size). Because batches are acknowledged as a group, a DataReader that cannot process an entire batch will lose the remaining DDS samples in it.
- 7.5.21 RELIABILITY QosPolicy on page 447
- 7.5.22 RESOURCE_LIMITS QosPolicy on page 451
7.5.13 LATENCYBUDGET QoS Policy

7.5.12.4 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.12.5 System Resource Considerations

While this QosPolicy does not directly affect the system resources used by Connext DDS, the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 that must be used in conjunction with the 7.5.12 HISTORY QosPolicy on page 420 will affect the amount of memory that Connext DDS will allocate for a DataWriter or DataReader.

7.5.13 LATENCYBUDGET QoS Policy

This QosPolicy can be used by a DDS implementation to change how it processes and sends data that has low latency requirements. The DDS specification does not mandate whether or how this parameter is used. Connext DDS uses it to prioritize the sending of asynchronously published data; see 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345.

This QosPolicy also applies to Topics. The Topic’s setting for the policy is ignored unless you explicitly make the DataWriter use it.

It contains the single member listed in Table 7.55 DDS_LatencyBudgetQosPolicy.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>duration</td>
<td>Provides a hint as to the maximum acceptable delay from the time the data is written to the time it is received by the subscribing applications.</td>
</tr>
</tbody>
</table>

7.5.13.1 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.14 LIFESPAN QoS Policy

The purpose of this QoS is to avoid delivering stale data to the application by specifying how long the data written by a DataWriter is considered valid.
Each data sample written by a `DataWriter` has an associated expiration time beyond which the data should not be delivered to any application. Once the sample expires, the data will be removed from the `DataWriter` and `DataReader` caches.

The expiration time of each sample from the `DataWriter's` cache is computed by adding the duration specified by this QoS policy to the time when the sample is added to the `DataWriter's` cache. This timestamp is not necessarily equal to the sample's source timestamp that can be provided by the user using the `DataWriter's` `write_w_timestamp()` or `write_w_params()` APIs.

The expiration time of each sample from the `DataReader's` cache is computed by adding the duration to the reception timestamp.

The Lifespan QosPolicy can be used to control how much data is stored by `Connext DDS`. Even if it is configured to store "all" of the data sent or received for a topic (see the 7.5.12 HISTORY QosPolicy on page 420), the total amount of data it stores may be limited by the Lifespan QosPolicy.

You may also use the Lifespan QosPolicy to ensure that applications do not receive or act on data, commands or messages that are too old and have "expired."

It includes the single member listed in Table 7.56 DDS_LifespanQosPolicy. For the default and valid range, please refer to the API Reference HTML documentation.

Table 7.56 DDS_LifespanQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>duration</td>
<td>Maximum duration for the data’s validity.</td>
</tr>
</tbody>
</table>

Although you can set the LIFESPAN QosPolicy on Topics, its value can only be used to initialize the LIFESPAN QosPolicies of `DataWriters`. The Topic’s setting for this QosPolicy does not directly affect the operation of `Connext DDS`, see 5.1.3 Setting Topic QosPolicies on page 219.

7.5.14.1 Properties

This QoS policy can be modified after the entity is enabled.

It does not apply to `DataReaders`, so there is no requirement that the publishing and subscribing sides use compatible values.

7.5.14.2 Related QoS Policies

- 7.5.2 BATCH QosPolicy (DDS Extension) on page 374 Be careful when configuring a `DataWriter` with a Lifespan `duration` shorter than the batch flush period (batch_flush_delay). If the batch does not fill up before the flush period elapses, the short `duration` by default will cause the DDS samples to be dropped without being sent. (You can, however, keep track of the number of these dropped samples via `writer_removed_batch_sample_dropped_sample_count` in the
7.5.15 LIVELINESS QosPolicy

8.3.7.2 DATA_READER_CACHE_STATUS on page 517. You can also choose not to drop these samples at all by setting the property dds.data_reader.accept_writer_removed_batch_samples to TRUE (by default it is set to FALSE); you can set this property via the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.

- 7.5.9 DURABILITY QosPolicy on page 411

7.5.14.3 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287

7.5.14.4 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

7.5.15 LIVELINESS QosPolicy

The LIVELINESS QosPolicy specifies how Connext DDS determines whether a DataWriter is “alive.” A DataWriter’s liveness is used in combination with the 7.5.17 OWNERSHIP QosPolicy on page 434 to maintain ownership of an instance (note that the 7.5.7 DEADLINE QosPolicy on page 405 is also used to change ownership when a DataWriter is still alive). That is, for a DataWriter to own an instance, the DataWriter must still be alive as well as honoring its DEADLINE contract.

It includes the members in Table 7.57 DDS_LivelinessQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

**Table 7.57 DDS_LivelinessQosPolicy**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
</table>
Setting a `DataWriter`'s kind of LIVELINESS specifies the mechanism that will be used to assert liveliness for the `DataWriter`. The `DataWriter`'s lease_duration then specifies the maximum period at which packets that indicate that the `DataWriter` is still alive are sent to matching `DataReaders`.

The various mechanisms are:

- **DDS_AUTOMATIC_LIVELINESS_QOS:**
  
  The `DomainParticipant` is responsible for automatically sending packets to indicate that the `DataWriter` is alive; this will be done at the rate determined by the assertions_per_lease_duration and lease_duration values. This setting is appropriate when the primary failure mode is that the publishing application itself dies. It does not cover the case in which the application is still alive but in an erroneous state—allowing the `DomainParticipant` to continue to assert liveliness for the `DataWriter` but preventing threads from calling write() on the `DataWriter`.
  
  As long as the internal threads spawned by Connext DDS for a `DomainParticipant` are running, then the liveliness of the `DataWriter` will be asserted regardless of the state of the rest of the application.
  
  This setting is certainly the most convenient, if the least accurate, method of asserting liveliness for a `DataWriter`.

- **DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS:**
  
  Connext DDS will assume that as long as the user application has asserted the liveliness of at least one `DataWriter` belonging to the same `DomainParticipant` or the liveliness of the `DomainParticipant` itself, then this `DataWriter` is also alive.

### Table 7.57 DDS_LivelinessQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>lease_duration</td>
<td>The timeout by which liveliness must be asserted for the <code>DataWriter</code> or the <code>DataWriter</code> will be considered inactive or not alive. Additionally, for <code>DataReaders</code>, the lease_duration also specifies the maximum period at which Connext DDS will check to see if the matching <code>DataWriter</code> is still alive.</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>A DataReader will consider a DataWriter not alive if the DataWriter does not assert its liveliness within the DataWriter's lease_duration, not the DataReader's lease_duration.</strong></td>
</tr>
<tr>
<td>DDS_Long</td>
<td>assertions_per_lease_duration</td>
<td>The number of assertions a <code>DataWriter</code> will send during a lease_duration period. This field only applies to <code>DataWriters</code> using DDS_AUTOMATIC_LIVELINESS_QOS kind and it is not considered during QoS compatibility checks. The default value is 3. A higher value will make the liveliness mechanism more robust against packet losses, but it will also increase the network traffic.</td>
</tr>
</tbody>
</table>
This setting allows the user code to control the assertion of liveliness for an entire group of DataWriters with a single operation on any of the DataWriters or their DomainParticipant. It's a good balance between control and convenience.

- **DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS:**

  The DataWriter is considered alive only if the user application has explicitly called operations that assert the liveliness for that particular DataWriter.

  This setting forces the user application to assert the liveliness for a DataWriter which gives the user application great control over when other applications can consider the DataWriter to be inactive, but at the cost of convenience.

With the MANUAL_BY_TOPIC, PARTICIPANT settings, user application code can assert the liveliness of DataWriters either explicitly by calling the assert_liveliness() operation on the DataWriter (as well as the DomainParticipant for the MANUAL_BY_PARTICIPANT setting) or implicitly by calling write() on the DataWriter. If the application does not use either of the methods mentioned at least once every lease_duration, then the subscribing application may assume that the DataWriter is no longer alive. Sending data MANUAL_BY_TOPIC will cause an assert message to be sent between the DataWriter and its matched DataReaders.

Publishing applications will monitor their DataWriters to make sure that they are honoring their LIVELINESS QosPolicy by asserting their liveliness at least at the period set by the lease_duration. If Connext DDS finds that a DataWriter has failed to have its liveliness asserted by its lease_duration, an internal thread will modify the DataWriter’s LIVELINESS LOST_STATUS and trigger its on_liveliness_lost() DataWriterListener callback if a listener exists, see 4.4 Listeners on page 189.

Setting the DataReader’s kind of LIVELINESS requests a specific mechanism for the publishing application to maintain the liveliness of DataWriters. The subscribing application may want to know that the publishing application is explicitly asserting the liveliness of the matching DataWriter rather than inferring its liveliness through the liveliness of its DomainParticipant or its sibling DataWriters.

The DataReader’s lease_duration specifies the maximum period at which matching DataWriters must have their liveliness asserted. In addition, in the subscribing application Connext DDS uses an internal thread that wakes up at the period set by the DataReader’s lease_duration to see if the DataWriter’s lease_duration has been violated.

When a matching DataWriter is determined to be dead (inactive), Connext DDS will modify the LIVELINESS_CHANGED_STATUS of each matching DataReader and trigger that DataReader’s on_liveliness_changed() DataReaderListener callback (if a listener exists).

Although you can set the LIVELINESS QosPolicy on Topics, its value can only be used to initialize the LIVELINESS QosPolicies of either a DataWriter or DataReader. It does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.
For more information on Liveliness, see 15.3.1.2 Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT on page 779.

### 7.5.15.1 Timing Considerations for MANUAL_BY_PARTICIPANT

As mentioned in 7.5.15.6 System Resource Considerations on page 432, a per-DomainParticipant internal Connext DDS thread will wake up periodically to check the liveliness of the DataWriters. This same thread also checks if any of the MANUAL_BY_PARTICIPANT DataWriters within the DomainParticipant have asserted liveliness (by calling either the DomainParticipant's or the DataWriter's assert_liveliness() API): if that is the case, that thread is responsible for sending a liveliness assertion message to any matched remote DomainParticipants.

The period of this thread is half of the time given by the minimum lease_duration across all of the MANUAL_BY_PARTICIPANT DataWriters within the DomainParticipant. As a result, an application asserting MANUAL_BY_PARTICIPANT DataWriters' liveliness at a period equal to or higher than half of the minimum configured lease_duration may run into local and/or remote DataWriter lost events.

For example, for a minimum lease_duration across all DataWriters of 2 seconds and an application assertion period (the time between calls to assert_liveliness()) of 1 second, the maximum time between two liveliness assertions will be given by:

```plaintext
"maximum time between two liveliness assertions" ~= ("minimum lease_duration across manual_by_participant datawriters" / 2) + "application assertion period"
```

which results in:

```plaintext
"maximum time between two liveliness assertions" ~= (2 / 2) second + 1 second = 2 seconds
```

This configuration results in potential (local and remote) liveliness losses since the lease_duration is close to the maximum time between two liveliness assertions (in fact, the situation will be a bit worse since this configuration has not accounted for network delay/jittering delivering liveliness assertion messages).

Therefore, to avoid unexpected liveliness losses, the user application should make sure that DataWriters' liveliness is asserted at a period that is shorter than half of the minimum lease duration. For additional information and diagrams, see 15.3.1.2 Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT on page 779.

### 7.5.15.2 Example

You can use LIVELINESS QosPolicy during system integration to ensure that applications have been coded to meet design specifications. You can also use it during run time to detect when systems are performing outside of design specifications. Receiving applications can take appropriate actions in response to disconnected DataWriters.

The LIVELINESS QosPolicy can be used to manage fail-over when the 7.5.17 OWNERSHIP QosPolicy on page 434 is set to EXCLUSIVE. This implies that the DataReader will only receive data
from the highest strength DataWriter that is alive (active). When that DataWriter’s liveliness expires, then Connext DDS will start delivering data from the next highest strength DataWriter that is still alive.

7.5.15.3 Properties

This QosPolicy cannot be modified after the Entity has been enabled.

The DataWriter and DataReader must use compatible settings for this QosPolicy. To be compatible, both of the following conditions must be true:

The DataWriter and DataReader must use one of the valid combinations shown in Table 7.58 Valid Combinations of Liveliness ‘kind’.

DataWriter’s lease_duration \(\leq\) DataReader’s lease_duration.

If this QosPolicy is found to be incompatible, the ON_OFFERED_INCOMPATIBLE_QOS and ON_REQUESTED_INCOMPATIBLE_QOS statuses will be modified and the corresponding Listeners called for the DataWriter and DataReader respectively.

Table 7.58 Valid Combinations of Liveliness ‘kind’

<table>
<thead>
<tr>
<th>DataWriter offers:</th>
<th>DataReader requests:</th>
<th>MANUAL_BY_TOPIC</th>
<th>MANUAL_BY_PARTICIPANT</th>
<th>AUTOMATIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANUAL_BY_TOPIC</td>
<td>compatible</td>
<td>compatible</td>
<td>compatible</td>
<td></td>
</tr>
<tr>
<td>MANUAL_BY_PARTICIPANT</td>
<td>incompatible</td>
<td>compatible</td>
<td>compatible</td>
<td></td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>incompatible</td>
<td>incompatible</td>
<td>compatible</td>
<td></td>
</tr>
</tbody>
</table>

7.5.15.4 Related QosPolicies

- 7.5.7 DEADLINE QosPolicy on page 405
- 7.5.17 OWNERSHIP QosPolicy on page 434
- 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438

7.5.15.5 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504
7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension)

7.5.16.6 System Resource Considerations

An internal thread in Connext DDS will wake up periodically to check the liveness of all the DataWriters. This happens both in the application that contains the DataWriters at the lease_duration set on the DataWriters as well as the applications that contain the DataReaders at the lease_duration set on the DataReaders. Therefore, as lease_duration becomes smaller, more CPU will be used to wake up threads and perform checks. A short lease_duration (or a high assertions_per_lease_duration) set on DataWriters may also use more network bandwidth because liveness packets are being sent at a higher rate—this is especially true when LIVELINESS kind is set to AUTOMATIC.

7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension)

This QosPolicy is used to partition the data published by a DataWriter across multiple channels. A channel is defined by a filter expression and a sequence of multicast locators.

By using this QosPolicy, a DataWriter can be configured to send data to different multicast groups based on the content of the data. Using syntax similar to those used in Content-Based Filters, you can associate different multicast addresses with filter expressions that operate on the values of the fields within the data. When your application’s code calls write(), data is sent to any multicast address for which the data passes the filter.

See Multi-channel DataWriters (Chapter 20 on page 896) for complete documentation on multi-channel DataWriters.

Note: Durable writer history is not supported for multi-channel DataWriters; an error is reported if a multi-channel DataWriter tries to configure Durable Writer History.

This QosPolicy includes the members presented in Table 7.59 DDS_MultiChannelQosPolicy, Table 7.60 DDS_ChannelSettings_t, and Table 7.61 DDS_TransportMulticastSettings_t. For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 7.59 DDS_MultiChannelQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_ChannelSettingsSeq</td>
<td>channels</td>
<td>A sequence of channel settings used to configure the channels' properties. If the length of the sequence is zero, the QosPolicy will be ignored. See Table 7.60 DDS_ChannelSettings.t.</td>
</tr>
<tr>
<td>char*</td>
<td>filter_name</td>
<td>Name of the filter class used to describe the filter expressions(^1). The following values are supported: DDS_SQLFILTER_NAME (see 5.4.6 SQL Filter Expression Notation on page 236) DDS_STRINGMATCHFILTER_NAME (see 5.4.7 STRINGMATCH Filter Expression Notation on page 245) More details are described in the API Reference HTML documentation.</td>
</tr>
</tbody>
</table>

\(^1\) In Java and C#, you can access the names of the built-in filters by using DomainParticipant.SQLFILTER_NAME and DomainParticipant.STRINGMATCHFILTER_NAME.
### Table 7.60 DDS_ChannelSettings_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TransportMulticastSettingsSeq</td>
<td>multicast_settings</td>
<td>A sequence of multicast settings used to configure the multicast addresses associated with a channel. <strong>The sequence cannot be empty.</strong> The maximum number of multicast locators in a channel is limited by default to 16. This is a hard limit that cannot be increased. However, this limit can be decreased by configuring the DomainParticipant property <code>dds.domainParticipant.max_announced_locator_list_size</code>. See Table 7.61 DDS_TransportMulticastSettings_t.</td>
</tr>
<tr>
<td>char*</td>
<td>filter_expression</td>
<td>A logical expression used to determine the data that will be published in the channel. <strong>This string cannot be NULL.</strong> An empty string always evaluates to TRUE. See 5.4.6 SQL Filter Expression Notation on page 236 and 5.4.7 STRINGMATCH Filter Expression Notation on page 245 for expression syntax.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>priority</td>
<td>A positive integer designating the relative priority of the channel, used to determine the transmission order of pending transmissions. Larger numbers have higher priority. To use publication priorities, the DataWriter’s 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444 must be set for asynchronous publishing and the DataWriter must use a FlowController that is configured for highest-priority-first (HPF) scheduling. See 7.6.4 Prioritized DDS Samples on page 476.</td>
</tr>
</tbody>
</table>

### Table 7.61 DDS_TransportMulticastSettings_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_StringSeq</td>
<td>transports</td>
<td>A sequence of transport aliases that specifies which transport should be used to publish multicast messages for this channel.</td>
</tr>
<tr>
<td>char*</td>
<td>receive_address</td>
<td>A multicast group address on which DataReaders subscribing to this channel will receive data.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>receive_port</td>
<td>The multicast port on which DataReaders subscribing to this channel will receive data.</td>
</tr>
</tbody>
</table>

The format of the **filter_expression** should correspond to one of the following filter classes:

- DDS_SQLFILTER_NAME (see 5.4.6 SQL Filter Expression Notation on page 236)
- DDS_STRINGMATCHFILTER_NAME (see 5.4.7 STRINGMATCH Filter Expression Notation on page 245)

A DataReader can use the ContentFilteredTopic API (see 5.4.5 Using a ContentFilteredTopic on page 233) to subscribe to a subset of the channels used by a DataWriter.
7.5.16.1 Example

See Multi-channel DataWriters (Chapter 20 on page 896).

7.5.16.2 Properties

This QosPolicy cannot be modified after the DataWriter is created.

It does not apply to DataReaders, so there is no requirement that the publishing and subscribing sides use compatible values.

7.5.16.3 Related Qos Policies

- 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654

7.5.16.4 Applicable Entities

- 7.3 DataWriters on page 287

7.5.16.5 System Resource Considerations

The following fields in the 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654 configure the resources associated with the channels stored in the MULTI_CHANNEL QosPolicy:

- channel_seq_max_length
- channel_filter_expression_max_length

For information about partitioning topic data across multiple channels, please refer to Multi-channel DataWriters (Chapter 20 on page 896).

7.5.17 OWNERSHIP QosPolicy

The OWNERSHIP QosPolicy specifies whether a DataReader receives data for an instance of a Topic sent by multiple DataWriters.

For non-keyed Topics, there is only one instance of the Topic.

This policy includes the single member shown in Table 7.62 DDS_OwnershipQosPolicy.
Table 7.62 DDS_OwnershipQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_OwnershipQosPolicyKind</td>
<td>kind</td>
<td>(default) DDS_SHARED_OWNERSHIP_QOS or DDS_EXCLUSIVE_OWNERSHIP_QOS</td>
</tr>
</tbody>
</table>

The **kind** of OWNERSHIP can be set to one of two values:

- **SHARED Ownership**
  
  When OWNERSHIP is **SHARED**, and multiple *DataWriters* for the *Topic* publishes the value of the same instance, all the updates are delivered to subscribing *DataReaders*. So in effect, there is no “owner;” no single *DataWriter* is responsible for updating the value of an instance. The subscribing application will receive modifications from all *DataWriters*.

- **EXCLUSIVE Ownership**
  
  When OWNERSHIP is **EXCLUSIVE**, each instance can only be owned by one *DataWriter* at a time. This means that a single *DataWriter* is identified as the exclusive owner whose updates are allowed to modify the value of the instance for matching *DataReaders*. Other *DataWriters* may submit modifications for the instance, but only those made by the current owner are passed on to the *DataReaders*. If a non-owner *DataWriter* modifies an instance, no error or notification is made; the modification is simply ignored. The owner of the instance can change dynamically.

  Note for non-keyed *Topics*, **EXCLUSIVE** ownership implies that *DataReaders* will pay attention to only one *DataWriter* at a time because there is only a single instance. For keyed *Topics*, *DataReaders* may actually receive data from multiple *DataWriters* when different *DataWriters* own different instances of the *Topic*.

This QosPolicy is often used to help users build systems that have redundant elements to safeguard against component or application failures. When systems have active and hot standby components, the Ownership QosPolicy can be used to ensure that data from standby applications are only delivered in the case of the failure of the primary.

The Ownership QosPolicy can also be used to create data channels or topics that are designed to be taken over by external applications for testing or maintenance purposes.

Although you can set the OWNERSHIP QosPolicy on *Topics*, its value can only be used to initialize the OWNERSHIP QosPolicies of either a *DataWriter* or *DataReader*. It does not directly affect the operation of *Connext DDS*, see 5.1.3 Setting Topic QosPolicies on page 219.
7.5.17 OWNERSHIP QosPolicy

7.5.17.1 How Connext DDS Selects which DataWriter is the Exclusive Owner

When OWNERSHIP is EXCLUSIVE, the owner of an instance at any given time is the DataWriter with the highest 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438 that is “alive” as defined by the 7.5.15 LIVELINESS QosPolicy on page 427 and has not violated the 7.5.7 DEADLINE QosPolicy on page 405 of the DataReader. OWNERSHIP_STRENGTH is simply an integer set by the DataWriter.

If the Topic’s data type is keyed (see 2.4 DDS Samples, Instances, and Keys on page 18), EXCLUSIVE ownership is determined on a per-instance basis. That is, the DataWriter owner of each instance is considered separately. A DataReader can receive values written by a lower strength DataWriter as long as those values are for instances that are not being written by a higher-strength DataWriter.

If there are multiple DataWriters with the same OWNERSHIP_STRENGTH writing to the same instance, Connext DDS resolves the tie by choosing the DataWriter with the smallest GUID (Globally Unique Identifier, see 15.1.1 Simple Participant Discovery on page 765.). This means that different DataReaders (in different applications) of the same Topic will all choose the same DataWriter as the owner when there are multiple DataWriters with the same strength.

The owner of an instance can change when:

- A DataWriter with a higher OWNERSHIP_STRENGTH publishes a value for the instance.
- The OWNERSHIP_STRENGTH of the owning DataWriter is dynamically changed to be less than the strength of an existing DataWriter of the instance.
- The owning DataWriter stops asserting its LIVELINESS (the DataWriter dies).
- The owning DataWriter violates the DEADLINE QosPolicy by not updating the value of the instance within the period set by the DEADLINE.

Note however, the change of ownership is not synchronous across different DataReaders in different participants. That is, DataReaders in different applications may not determine that the ownership of an instance has changed at exactly the same time.

7.5.17.2 Example

OWNERSHIP is really a property that is shared between DataReaders and DataWriters of a Topic. However, in a system, some Topics will be exclusively owned and others will be shared. System requirements will determine which are which.

An example of a Topic that may be shared is one that is used by applications to publish alarm messages. If the application detects an anomalous condition, it will use a DataWriter to write a Topic “Alarm.” Another application that records alarms into a system log file will have a DataReader that subscribes to “Alarm.” In this example, any number of applications can publish the “Alarm” message. There is no concept that only one application at a time is allowed to publish the “Alarm” message, so in this case, the OWNERSHIP of the DataWriters and DataReaders should be set to SHARED.
In a different part of the system, EXCLUSIVE OWNERSHIP may be used to implement redundancy in support of fault tolerance. Say, the distributed system controls a traffic system. It monitors traffic and changes the information posted on signs, the operation of metering lights, and the timing of traffic lights. This system must be tolerant to failure of any part of the system including the application that actually issues commands to change the lights at a particular intersection.

One way to implement fault tolerance is to create the system redundantly both in hardware and software. So if a piece of the running system fails, a backup can take over. In systems where failover from the primary to backup system must be seamless and transparent, the actual mechanics of failover must be fast, and the redundant component must immediately pickup where the failed component left off. For the network connections of the component, Connext DDS can provided redundant DataWriter and DataReaders.

In this case, you would not want the DataReaders to receive redundant messages from the redundant DataWriters. Instead you will want the DataReaders to only receive messages from the primary application and only from a backup application when a failure occurs. To continue our example, if we have redundant applications that all try to control the lights at an intersection, we would want the DataReaders on the light to receive messages only from the primary application. To do so, we should configure the DataWriters and DataReaders to have EXCLUSIVE OWNERSHIP and set the OWNERSHIP_STRENGTH differently on different redundant applications to distinguish between primary and backup systems.

7.5.17.3 Properties

This QosPolicy cannot be modified after the Entity is enabled.

It must be set to the same kind on both the publishing and subscribing sides. If a DataWriter and DataReader of the same topic are found to have different kinds set for the OWNERSHIP QoS, the ON_OFFERED_INCOMPATIBLE_QOS and ON_REQUESTED_INCOMPATIBLE_QOS statuses will be modified and the corresponding Listeners called for the DataWriter and DataReader respectively.

7.5.17.4 Related QosPolicies

- 7.5.7 DEADLINE QosPolicy on page 405
- 7.5.15 LIVELINESS QosPolicy on page 427
- 7.5.18 OWNERSHIP_STRENGTH QosPolicy on the next page

7.5.17.5 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504
7.5.17.6 System Resource Considerations

This QosPolicy does not significantly impact the use of system resources.

7.5.18 OWNERSHIP_STRENGTH QosPolicy

The OWNERSHIP_STRENGTH QosPolicy is used to rank DataWriters of the same instance of a Topic, so that Connext DDS can decide which DataWriter will have ownership of the instance when the 7.5.17 OWNERSHIP QosPolicy on page 434 is set to EXCLUSIVE.

It includes the member in Table 7.63 DDS_OwnershipStrengthQosPolicy. For the default and valid range, please refer to the API Reference HTML documentation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>value</td>
<td>The strength value used to arbitrate among multiple DataWriters.</td>
</tr>
</tbody>
</table>

This QosPolicy only applies to DataWriters when EXCLUSIVE OWNERSHIP is used. The strength is simply an integer value, and the DataWriter with the largest value is the owner. A deterministic method is used to decide which DataWriter is the owner when there are multiple DataWriters that have equal strengths. See 7.5.17.1 How Connext DDS Selects which DataWriter is the Exclusive Owner on page 436 for more details.

7.5.18.1 Example

Suppose there are two DataWriters sending DDS samples of the same Topic instance, one as the main DataWriter, and the other as a backup. If you want to make sure the DataReader always receive from the main one whenever possible, then set the main DataWriter to use a higher ownership_strength value than the one used by the backup DataWriter.

7.5.18.2 Properties

This QosPolicy can be changed at any time.

It does not apply to DataReaders, so there is no requirement that the publishing and subscribing sides use compatible values.

7.5.18.3 Related QosPolicies

- 7.5.17 OWNERSHIP QosPolicy on page 434
7.5.19 PROPERTY QosPolicy (DDS Extension)

7.5.18.4 Applicable Entities

- 7.3 DataWriters on page 287

7.5.18.5 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

7.5.19 PROPERTY QosPolicy (DDS Extension)

The PROPERTY QosPolicy stores name/value (string) pairs that can be used to configure certain parameters of Connext DDS that are not exposed through formal QoS policies.

It can also be used to store and propagate application-specific name/value pairs that can be retrieved by user code during discovery. This is similar to the USER_DATA QosPolicy, except this policy uses (name, value) pairs, and you can select whether or not a particular pair should be propagated (included in the built-in topic). By default, properties are not propagated during discovery.

It includes the member in Table 7.64 DDS_PropertyQosPolicy.

Table 7.64 DDS_PropertyQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_PropertySeq</td>
<td>value</td>
<td>A sequence of (name, value) pairs and booleans that indicate whether the pair should be propagated (included in the entity’s built-in topic upon discovery).</td>
</tr>
</tbody>
</table>

The Property QoS stores name/value pairs for an Entity. Both the name and value are strings. Certain configurable parameters for Entities that do not have a formal DDS QoS definition may be configured via this QoS by using a predefined name and the desired setting in string form.

You can find a complete list of predefined properties in the Property Reference Guide.

You can manipulate the sequence of properties (name, value pairs) with the standard methods available for sequences. You can also use the helper class, DDSPropertyQosPolicyHelper, which provides another way to work with a PropertyQosPolicy object.

The PropertyQosPolicy may be used to configure:

- Durable writer history (see 13.3.2 How To Configure Durable Writer History on page 741)
- Durable reader state (see 13.4.4 How To Configure a DataReader for Durable Reader State on page 747)
7.5.19 PROPERTY QosPolicy (DDS Extension)

- Built-in and extension Transport Plugins (see 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy on page 801, 42.2 Setting Up a Transport with the Property QoS on page 1123, Configuring the TCP Transport (Chapter 44 on page 1148)).

- Automatic registration of built-in types (see 3.2.1 Registering Built-in Types on page 39)

- 9.6 Clock Selection on page 677

- 7.3.18 Turbo Mode and Automatic Throttling for DataWriter Performance—Experimental Features on page 344

- Location or content of your license from RTI (see License Management, in the RTI Connext DDS Installation Guide)

In addition, you can add your own name/value pairs to the Property QoS of an Entity. Start them with a prefix other than dds., com.rti., or rti., so that they do not fail validation. (See 7.5.19.1 Property Validation on the next page.) You may also use this QosPolicy to direct Connext DDS to propagate these name/value pairs with the discovery information for the Entity. Applications that discover the Entity can then access the user-specific name/value pairs in the discovery information of the remote Entity. This allows you to add meta-information about an Entity for application-specific use, for example, authentication/authorization certificates (which can also be done using the User or Group Data QoS).

Reasons for using the PropertyQosPolicy include:

- Some features can only be configured through the PropertyQosPolicy, not through other QoS or APIs (for example, Durable Reader State, Durable Writer History, Built-in Types, Monotonic Clock).

- Alternative way to configure built-in transports settings. For example, to use non-default values for the built-in transports without using the PropertyQosPolicy, you would have to create a DomainParticipant disabled, change the built-in transport property settings, then enable the DomainParticipant. Using the PropertyQosPolicy to configure built-in transport settings will save you the work of enabling and disabling the DomainParticipant. Also, transport settings are not a QoS and therefore cannot be configured through an XML file. By configuring built-in transport settings through the PropertyQosPolicy instead, XML files can be used.

When using the Java or .NET APIs, transport configuration must take place through the PropertyQosPolicy (not through the transport property structures).
• Alternative way to support multiple instances of built-in transports (without using Transport API).
• Alternative way to dynamically load extension transports (such as RTI Secure WAN Transport\(^1\) or RTI TCP Transport\(^2\)) or user-created transport plugins in C/C++ language bindings. If the extension or user-created transport plugin is installed using the transport API instead, the library that extra transport library/code will need to be linked into your application and may require recompilation.
• Allows full pluggable transport configuration for non-C/C++ language bindings (Java, C++/CLI, C#, etc.) The pluggable transport API is not available in those languages. Without using PropertyQosPolicy, you cannot use extension transports (such as RTI Secure WAN Transport) and you cannot create your own custom transport.
• Alternative way to provide a license for platforms that do not support a file system, or if a default license location is not feasible and environment variables are not supported.

The PropertyQosPolicyHelper operations are described in Table 7.65 PropertyQoSPolicyHelper Operations. For more information, see the API Reference HTML documentation.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_number_of_properties</td>
<td>Gets the number of properties in the input policy.</td>
</tr>
<tr>
<td>assert_property</td>
<td>Asserts the property identified by name in the input policy. (Either adds it, or replaces an existing one.)</td>
</tr>
<tr>
<td>add_property</td>
<td>Adds a new property to the input policy.</td>
</tr>
<tr>
<td>assert_pointer_property</td>
<td>Asserts the property identified by name in the input policy. Used when the property to store is a pointer.</td>
</tr>
<tr>
<td>add_pointer_property</td>
<td>Adds a new property to the input policy. Used when the property to store is a pointer.</td>
</tr>
<tr>
<td>lookup_property</td>
<td>Searches for a property in the input policy given its name.</td>
</tr>
<tr>
<td>remove_property</td>
<td>Removes a property from the input policy.</td>
</tr>
<tr>
<td>get_properties</td>
<td>Retrieves a list of properties whose names match the input prefix.</td>
</tr>
</tbody>
</table>

### 7.5.19.1 Property Validation

All the properties that Connext DDS provides (which begin with dds., com.rti., or rti.) are validated when the entity or the plugin is created. This validation is done to avoid using an unknown or incorrect property

---

\(^1\)RTI Secure WAN Transport is an optional component that is installed separately.

\(^2\)RTI TCP Transport is included with your Connext DDS distribution but is not a built-in transport and therefore not enabled by default.
name (for example, due to a typo). Without this validation, Connext DDS ignores the unknown property name, and you might not know why the property's configuration isn't being applied.

By default, at the creation of an entity or a plugin, if you specify an incorrect property name, Connext DDS logs an exception similar to the following:

- Entity:

```
DDS_PropertyQosPolicy_validatePropertyNames:Unexpected property: dds.type_consistency.ignore_sequence_bounds. Closest valid property: dds.type_consistency.ignore_sequence_bounds
DDS_DataReaderQos_is_consistentI:inconsistent QoS property
DDS_Subscriber_create_datreader_disabledI:ERROR: Inconsistent QoS
```

- Plugin, such as TCPv4:

```
NDDS_Transport_TCPv4_create:!get transport TCPv4 plugin property from DDS Property
```

You can configure the behavior of this validation by setting a property at the DomainParticipant level. The DomainParticipant's DataWriters and DataReaders use the participant's setting. Or you can set the property at the plugin level.

- At the entity-level setting, you can set the property `dds.participant.property_validation_action` to any of the following options:

  - (default) `VALIDATION_ACTION_EXCEPTION`: validate properties. Upon failure, log errors and fail.
  - `VALIDATION_ACTION_SKIP`: skip validation.
  - `VALIDATION_ACTION_WARNING`: validate properties. Upon failure, log warnings and do not fail.

- At the plugin level setting, you can set the property `<plugin_name>.property_validation_action`:

  - Options:

    - `VALIDATION_ACTION_EXCEPTION`: validate properties. Upon failure, log errors and fail.
    - `VALIDATION_ACTION_SKIP`: skip validation.
    - `VALIDATION_ACTION_WARNING`: validate properties. Upon failure, log warnings and do not fail.
• If the property is not set, the plugin property validation behavior will be the same as that of the DomainParticipant, which by default is VALIDATION_ACTION_EXCEPTION.

• For example, to set the property_validation_action for the dds.transport.TCPv4.tcp1 transport plugin via XML:

```xml
<domain_participant_qos>
  <property>
    <value>
      <element>
        <name>dds.transport.load_plugins</name>
        <value>dds.transport.TCPv4.tcp1</value>
      </element>
      <element>
        <name>dds.transport.TCPv4.tcp1.property_validation_action</name>
        <value>VALIDATION_ACTION_WARNING</value>
      </element>
    </value>
  </property>
</domain_participant_qos>
```

In general, it is recommended that you use dds.participant.property_validation_action to control the validation of the properties for both the Connext DDS core libraries and any plugins you might use. However, there are cases where you might want to configure different behaviors for the core libraries and the plugins. For example, if you are running a customized version of the plugins that supports a new, experimental property, you will need to disable the DomainParticipant validation via dds.participant.property_validation_action, but still keep the plugin validation (for example, dds.transport.TCPv4.tcp1.property_validation_action). Here's an example of disabling the DomainParticipant level validation and enabling a plugin level validation:

```xml
<domain_participant_qos>
  <property>
    <value>
      <element>
        <name>dds.participant.property_validation_action</name>
        <value>VALIDATION_ACTION_SKIP</value>
      </element>
      <element>
        <name>dds.transport.TCPv4.tcp1.property_validation_action</name>
        <value>VALIDATION_ACTION_EXCEPTION</value>
      </element>
    </value>
  </property>
</domain_participant_qos>
```

Note that the validation is sequential: first the property is validated when the DomainParticipant is created, then it is validated when the plugin is created. For example, consider that the DomainParticipant sets the property dds.participant.property_validation_action to VALIDATION_ACTION_EXCEPTION, but the plugin is configured to skip the unknown property. In this case, DomainParticipant creation will fail, and the plugin will never get created. As described above, if you are customizing the plugin, set the
validation to VALIDATION_ACTION_SKIP at the DomainParticipant level, then set the plugin property validation to VALIDATION_ACTION_EXCEPTION. By doing that, the properties will be validated just at the plugin level.

You can find a complete list of the Connext DDS predefined properties in the Property Reference Guide.

### 7.5.19.2 Properties

This QosPolicy can be changed at any time.

There is no requirement that the publishing and subscribing sides use compatible values.

### 7.5.19.3 Related QosPolicies

- 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654

### 7.5.19.4 Applicable Entities

- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504
- 9.3 DomainParticipants on page 609

### 7.5.19.5 System Resource Considerations

The 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654 contains several fields for configuring the resources associated with the properties stored in this QosPolicy.

### 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension)

This QosPolicy determines the DataWriter’s publishing mode, either asynchronous or synchronous.

The publishing mode controls whether data is written synchronously—in the context of the user thread when calling write(), or asynchronously—in the context of a separate thread internal to Connext DDS.

**Note:** For information on asynchronous DataWriters and sender-side filtering, see 5.4.2 Where Filtering is Applied—Publishing vs. Subscribing Side on page 228.

Each Publisher spawns a single asynchronous publishing thread (set in its 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345) to serve all its asynchronous DataWriters.

When data is written asynchronously, a FlowController (7.6 FlowControllers (DDS Extension) on page 470), identified by flow_controller_name, can be used to shape the network traffic. The FlowController's properties determine when the asynchronous publishing thread is allowed to send data and how much.
The fastest way for Connext DDS to send data is for the user thread to execute the middleware code that actually sends the data itself. However, there are times when user applications may need or want an internal middleware thread to send the data instead. For instance, for sending large data reliably, an asynchronous thread must be used (see 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345). See also 23.3 Large Data Fragmentation on page 970.

This QosPolicy can select a FlowController to prioritize or shape the data flow sent by a DataWriter to DataReaders. Shaping a data flow usually means limiting the maximum data rates with which the middleware will send data for a DataWriter. The FlowController will buffer data sent faster than the maximum rate by the DataWriter, and then only send the excess data when the user send rate drops below the maximum rate.

If kind is set to DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS, the flow controller referred to by flow_controller_name must exist. Otherwise, the setting will be considered inconsistent.

This QosPolicy includes the members in Table 7.66 DDS_PublishModeQosPolicy. For the defaults, please refer to the API Reference HTML documentation.

### Table 7.66 DDS_PublishModeQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_PublishMode QosPolicyKind</td>
<td>kind</td>
<td>Either:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_SYNCHRONOUS_PUBLISH_MODE_QOS</td>
</tr>
<tr>
<td>char*</td>
<td>flow_controller_name</td>
<td>Name of the associated flow controller.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>There are three built-in FlowControllers:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_DEFAULT_FLOW_CONTROLLER_NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_FIXED_RATE_FLOW_CONTROLLER_NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_ON_DEMAND_FLOW_CONTROLLER_NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>You may also create your own FlowControllers.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See 7.6 FlowControllers (DDS Extension) on page 470.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>priority</td>
<td>A positive integer designating the relative priority of the DataWriter, used to determine the transmission order of pending writes. To use publication priorities, this QosPolicy’s kind must be DDS_ASYNCHRONOUS_PUBLISH_MODE_QOS and the DataWriter must use a FlowController with a highest-priority first (HPF) scheduling_policy. See 7.6.4 Prioritized DDS Samples on page 476.</td>
</tr>
</tbody>
</table>

The maximum number of DDS samples that will be coalesced depends on NDDS_Transport_Property_t::gather_send_buffer_count_max (each DDS sample requires at least 2-4 gather-send buffers).
Performance can be improved by increasing \texttt{NDDS\_Transport\_Property\_t::gather\_send\_buffer\_count\_max}. Note that the maximum value is operating system dependent.

\textit{Connext DDS} queues DDS samples until they can be sent by the asynchronous publishing thread (as determined by the corresponding FlowController).

The number of DDS samples that will be queued is determined by the \texttt{7.5.12 HISTORY QosPolicy} on page 420: when using \texttt{KEEP\_LAST}, the most recent \texttt{depth} DDS samples are kept in the queue.

Once unsent DDS samples are removed from the queue, they are no longer available to the asynchronous publishing thread and will therefore never be sent.

Unless \texttt{flow\_controller\_name} points to one of the built-in FlowControllers, finalizing the DataWriterQos will also free the string pointed to by \texttt{flow\_controller\_name}. Therefore, you should use \texttt{DDS\_String\_dup()} before passing the string to \texttt{flow\_controller\_name}, or reset \texttt{flow\_controller\_name} to NULL before the destructing/finalizing the QoS.

\textbf{Advantages of Asynchronous Publishing:}

Asynchronous publishing may increase latency, but offers the following advantages:

- The \texttt{write()} call does not make any network calls and is therefore faster and more deterministic. This becomes important when the user thread is executing time-critical code.

- When data is written in bursts or when sending large data types as multiple fragments, a flow controller can throttle the send rate of the asynchronous publishing thread to avoid flooding the network.

- Asynchronously written DDS samples for the same destination will be coalesced into a single network packet which reduces bandwidth consumption.

\textbf{7.5.20.1 Properties}

This QosPolicy cannot be modified after the \textit{DataWriter} is created.

Since it is only for \textit{DataWriters}, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

\textbf{7.5.20.2 Related QosPolicies}

- \texttt{7.4.1 ASYNCHRONOUS\_PUBLISHER QosPolicy (DDS Extension)} on page 345

- \texttt{7.5.12 HISTORY QosPolicy} on page 420

\textbf{7.5.20.3 Applicable Entities}

- \texttt{7.3 DataWriters} on page 287
7.5.20.4 System Resource Considerations

See 7.5.22.1 Configuring Resource Limits for Asynchronous DataWriters on page 453.

System resource usage depends on the settings in the corresponding FlowController (see 7.6 FlowControllers (DDS Extension) on page 470).

7.5.21 RELIABILITY QosPolicy

This RELIABILITY QosPolicy determines whether or not data published by a DataWriter will be reliably delivered by Connext DDS to matching DataReaders. The reliability protocol used by Connext DDS is discussed in Reliable Communications (Chapter 11 on page 688).

The reliability of a connection between a DataWriter and DataReader is entirely user configurable. It can be done on a per DataWriter/DataReader connection. A connection may be configured to be "best effort" which means that Connext DDS will not use any resources to monitor or guarantee that the data sent by a DataWriter is received by a DataReader.

For some use cases, such as the periodic update of sensor values to a GUI displaying the value to a person, "best effort" delivery is often good enough. It is certainly the fastest, most efficient, and least resource-intensive (CPU and network bandwidth) method of getting the newest/latest value for a topic from DataWriters to DataReaders. But there is no guarantee that the data sent will be received. It may be lost due to a variety of factors, including data loss by the physical transport such as wireless RF or even Ethernet. Packets received out of order are dropped and reported as lost with the reason LOST_BY_WRITER (see 8.3.7.7 SAMPLE_LOST Status on page 527).

However, there are data streams (topics) in which you want an absolute guarantee that all data sent by a DataWriter is received reliably by DataReaders. This means that Connext DDS must check whether or not data was received, and repair any data that was lost by resending a copy of the data as many times as it takes for the DataReader to receive the data.

Connext DDS uses a reliability protocol configured and tuned by these QoS policies:

- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389
- 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570
- 7.5.22 RESOURCE_LIMITS QosPolicy on page 451

The Reliability QoS policy is simply a switch to turn on the reliability protocol for a DataWriter/DataReader connection. The level of reliability provided by Connext DDS is determined by the configuration of the aforementioned QoS policies.

You can configure Connext DDS to deliver ALL data in the order they were sent (also known as absolute or strict reliability). Or, as a trade-off for less memory, CPU, and network usage, you can choose a
reduced level of reliability where only the last N values are guaranteed to be delivered reliably to DataReaders (where N is user-configurable). With the reduced level of reliability, there are no guarantees that the data sent before the last N are received. Only the last N data packets are monitored and repaired if necessary.

It includes the members in Table 7.67 DDS_ReliabilityQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

### Table 7.67 DDS_ReliabilityQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_ReliabilityQosPolicyKind</td>
<td>kind</td>
<td>Can be either:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DDS_BESTEFFORT_RELIABILITY_QOS: DDS data samples are sent once and</td>
</tr>
<tr>
<td></td>
<td></td>
<td>missed samples are acceptable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DDS_RELIABLE_RELIABILITY_QOS: Connext DDS will make sure that data sent</td>
</tr>
<tr>
<td></td>
<td></td>
<td>is received and missed DDS samples are resent.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>max_blocking_time</td>
<td>How long a DataWriter can block on a write() when the send queue is full due</td>
</tr>
<tr>
<td></td>
<td></td>
<td>to un-acknowledged messages. (Has no meaning for DataReaders.)</td>
</tr>
<tr>
<td>DDS_ReliabilityQosPolicyAcknowledgmentModeKind</td>
<td>acknowledgment_kind</td>
<td>Kind of reliable acknowledgment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Only applies when kind is RELIABLE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Sets the kind of acknowledgments supported by a DataWriter and sent by</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DataReader.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DDS_PROTOCOL_Acknowledgment_MODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DDS_APPLICATION.Auto_Acknowledgment_MODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- DDS_APPLICATION.Explicit_Acknowledgment_MODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See 7.3.12.1 Application Acknowledgment Kinds on page 318</td>
</tr>
</tbody>
</table>

The kind of RELIABILITY can be either:

- **BEST EFFORT**
  - Connext DDS will send DDS data samples only once to DataReaders. No effort or resources are spent to track whether or not sent DDS samples are received. Minimal resources are used. This is the most deterministic method of sending data since there is no indeterministic delay that can be introduced by buffering or resending data. DDS data samples may be lost. This setting is good for periodic data.

- **RELIABLE**
  - Connext DDS will send DDS samples reliably to DataReaders—buffering sent data until they have been acknowledged as being received by DataReaders and resending any DDS samples that may
have been lost during transport. Additional resources configured by the HISTORY and RESOURCE_LIMITS QosPolicies may be used. Extra packets will be sent on the network to query (heartbeat) and acknowledge the receipt of DDS samples by the DataReader. This setting is a good choice when guaranteed data delivery is required; for example, sending events or commands.

To send large data reliably, you will also need to set the 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444 kind to DDSASYNCHRONOUS_PUBLISH_MODE_QOS. Large in this context means that the data size is larger than the transport message_size_max property value. See 23.3 Large Data Fragmentation on page 970.

While a DataWriter sends data reliably, the 7.5.12 HISTORY QosPolicy on page 420 and 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 determine how many DDS samples can be stored while waiting for acknowledgements from DataReaders. A DDS sample that is sent reliably is entered in the DataWriter’s send queue awaiting acknowledgement from DataReaders. How many DDS samples that the DataWriter is allowed to store in the send queue for a data-instance depends on the kind of the HISTORY QoS as well as the max_samples_per_instance and max_samples parameter of the RESOURCE_LIMITS QoS.

If the HISTORY kind is KEEP_LAST, then the DataWriter is allowed to have the HISTORY depth number of DDS samples per instance of the Topic in the send queue. Should the number of unacknowledged DDS samples in the send queue for a data-instance reach the HISTORY depth, then the next DDS sample written by the DataWriter for the instance will overwrite the oldest DDS sample for the instance in the queue. This implies that an unacknowledged DDS sample may be overwritten and thus lost. So even if the RELIABILITY kind is RELIABLE, if the HISTORY kind is KEEP_LAST, it is possible that some data sent by the DataWriter will not be delivered to the DataReader. What is guaranteed is that if the DataWriter stops writing, the last N DDS samples that the DataWriter wrote will be delivered reliably; where n is the value of the HISTORY depth.

However, if the HISTORY kind is KEEP_ALL, then when the send queue is filled with unacknowledged DDS samples (either due to the number of unacknowledged DDS samples for an instance reaching the RESOURCE_LIMITS max_samples_per_instance value or the total number of unacknowledged DDS samples have reached the size of the send queue as specified by RESOURCE_LIMITS max_samples), the next write() operation on the DataWriter will block until either a DDS sample in the queue has been fully acknowledged by DataReaders and thus can be overwritten or a timeout of RELIABILITY max_blocking_period has been reached.

If there is still no space in the queue when max_blocking_time is reached, the write() call will return a failure with the error code DDS_RETCODE_TIMEOUT.

Thus for strict reliability—a guarantee that all DDS data samples sent by a DataWriter are received by DataReaders—you must use a RELIABILITY kind of RELIABLE and a HISTORY kind of KEEP_ALL for both the DataWriter and the DataReader.
Although you can set the RELIABILITY QosPolicy on Topics, its value can only be used to initialize the RELIABILITY QosPolicies of either a DataWriter or DataReader. It does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.

### 7.5.21.1 Example

This QosPolicy is used to achieve reliable communications, which is discussed in Reliable Communications (Chapter 11 on page 688) and 11.3.1 Enabling Reliability on page 696.

### 7.5.21.2 Properties

This QosPolicy cannot be modified after the Entity has been enabled.

The DataWriter and DataReader must use compatible settings for this QosPolicy. To be compatible, the DataWriter and DataReader must use one of the valid combinations for the Reliability kind (see Table 7.68 Valid Combinations of Reliability ‘kind’), and one of the valid combinations for the acknowledgment_kind (see Table 7.69 Valid Combinations of Reliability ‘acknowledgment_kind’):

#### Table 7.68 Valid Combinations of Reliability ‘kind’

<table>
<thead>
<tr>
<th>DataWriter offers:</th>
<th>DataReader requests:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>BEST_EFFORT</td>
</tr>
<tr>
<td></td>
<td>RELIABLE</td>
</tr>
<tr>
<td>BEST_EFFORT</td>
<td>compatible</td>
</tr>
<tr>
<td>RELIABLE</td>
<td>compatible</td>
</tr>
<tr>
<td></td>
<td>incompatible</td>
</tr>
</tbody>
</table>

#### Table 7.69 Valid Combinations of Reliability ‘acknowledgment_kind’

<table>
<thead>
<tr>
<th>DataWriter offers:</th>
<th>DataReader requests:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>PROTOCOL</td>
</tr>
<tr>
<td></td>
<td>APPLICATION_AUTO</td>
</tr>
<tr>
<td></td>
<td>APPLICATION_EXPLICIT</td>
</tr>
<tr>
<td>PROTOCOL</td>
<td>compatible</td>
</tr>
<tr>
<td>APPLICATION_AUTO</td>
<td>compatible</td>
</tr>
<tr>
<td>APPLICATION_EXPLICIT</td>
<td>compatible</td>
</tr>
<tr>
<td></td>
<td>incompatible</td>
</tr>
<tr>
<td></td>
<td>compatible</td>
</tr>
<tr>
<td></td>
<td>compatible</td>
</tr>
</tbody>
</table>

If this QosPolicy is found to be incompatible, statuses ON_OFFERED_INCOMPATIBLE_QOS and ON_REQUESTED_INCOMPATIBLE_QOS will be modified and the corresponding Listeners called for the DataWriter and DataReader, respectively.
There are no compatibility issues regarding the value of `max_blocking_wait`, since it does not apply to `DataReaders`.

### 7.5.21.3 Related QosPolicies

- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444
- 7.5.22 RESOURCE_LIMITS QosPolicy below

### 7.5.21.4 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

### 7.5.21.5 System Resource Considerations

Setting the `kind` to `RELIABLE` will cause Connext DDS to use up more resources to monitor and maintain a reliable connection between a `DataWriter` and all of its reliable `DataReaders`. This includes the use of extra CPU and network bandwidth to send and process heartbeat, ACK/NACK, and repair packets (see Reliable Communications (Chapter 11 on page 688)).

Setting `max_blocking_time` to a non-zero number may block the sending thread when the RELIABILITY kind is `RELIABLE`.

### 7.5.22 RESOURCE_LIMITS QosPolicy

For the reliability protocol (and the 7.5.9 DURABILITY QosPolicy on page 411), this QosPolicy determines the actual maximum queue size when the 7.5.12 HISTORY QosPolicy on page 420 is set to KEEP_ALL.

In general, this QosPolicy is used to limit the amount of system memory that Connext DDS can allocate. For embedded real-time systems and safety-critical systems, pre-determination of maximum memory usage is often required. In addition, dynamic memory allocation could introduce non-deterministic latencies in time-critical paths.

It includes the members in Table 7.70 DDS_ResourceLimitsQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.
### Table 7.70 DDS_ResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_samples</td>
<td>Maximum number of live DDS samples that Connext DDS can store for a DataWriter/DataReader. This is a physical limit.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_instances</td>
<td>Maximum number of active instances that can be managed by a DataWriter/DataReader. (See 8.3.8.6.1 Active State and Minimum State on page 540.) For DataReaders, max_instances must be &lt;= max_total_instances in the 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576. See also: 7.5.22.2 Example on the next page.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_samples_per_in-</td>
<td>On a DataWriter, this resource limit represents the maximum number of DDS samples of any one instance that Connext DDS will store for a DataWriter. On a DataReader, this resource limit represents the maximum number of DDS samples of any one instance that are stored in the DataReader output queue—that is, the queue from which the application takes/reads samples. For keyed types and DataReaders, this value only applies to DDS samples with an instance state of DDS_ALIVE_INSTANCE_STATE. If a keyed Topic is not used, then max_samples_per_instance must equal max_samples. How this property behaves depends on your HISTORY and RELIABILITY QoS configurations. See 7.5.12 HISTORY QosPolicy on page 420.</td>
</tr>
<tr>
<td></td>
<td>stance</td>
<td></td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_samples</td>
<td>Initial number of DDS samples that Connext DDS will store for a DataWriter/DataReader. (DDS extension)</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_instances</td>
<td>Initial number of instances that can be managed by a DataWriter/DataReader. (DDS extension)</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>instance_hash_buckets</td>
<td>Number of hash buckets, which are used by Connext DDS to facilitate instance lookup. (DDS extension).</td>
</tr>
</tbody>
</table>

One of the most important fields is max_samples, which sets the size and causes memory to be allocated for the send or receive queues. For information on how this policy affects reliability, see 11.3.2 Tuning Queue Sizes and Other Resource Limits on page 696.

When a DataWriter orDataReader is created, the initial_instances and initial_samples parameters determine the amount of memory first allocated for the those Entities. As the application executes, if more space is needed in the send/receive queues to store DDS samples or as more instances are created, then Connext DDS will automatically allocate memory until the limits of max_instances and max_samples are reached.

You may set initial_instances = max_instances and initial_samples = max_samples if you do not want Connext DDS to dynamically allocate memory after initialization.

For keyed Topics, the max_samples_per_instance field in this policy represents the maximum number of DDS samples with the same key that are allowed to be stored by a DataWriter (in the DataWriter's queue) or by theDataReader (in the DataReader's output queue)—that is, the queue from which the application
takes/reads samples). The **max_samples_per_instance** field is a logical limit. The hard physical limit is determined by **max_samples**. However, because the theoretical number of instances may be quite large (as set by **max_instances**), you may not want Connext DDS to allocate the total memory needed to hold the maximum number of DDS samples per instance for all possible instances (**max_samples_per_instance *** max_instances**) because during normal operations, the application will never have to hold that much data for the Entity.

So it is possible that an Entity will hit the physical limit **max_samples** before it hits the **max_samples_per_instance** limit for a particular instance. However, Connext DDS must be able to store **max_samples_per_instance** for at least one instance. Therefore, **max_samples_per_instance** must be <= **max_samples**.

If a keyed data type is not used, there is only a single instance of the **Topic**, so **max_samples_per_instance must equal max_samples**.

Once a physical or logical limit is hit, then how Connext DDS deals with new DDS data samples being sent or received for a **DataWriter** or **DataReader** is described in the 7.5.12 HISTORY QosPolicy on page 420 setting of DDS_KEEP_ALL_HISTORY_QOS. It is closely tied to whether or not a reliable connection is being maintained.

Although you can set the RESOURCE_LIMITS QosPolicy on **Topics**, its value can only be used to initialize the RESOURCE_LIMITS QosPolicies of either a **DataWriter** or **DataReader**. It does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.

### 7.5.22.1 Configuring Resource Limits for Asynchronous DataWriters

When using an asynchronous **Publisher**, if a call to **write()** is blocked due to a resource limit, the block will last until the timeout period expires, which will prevent others from freeing the resource. To avoid this situation, make sure that the **DomainParticipant's outstanding_asynchronous_sample_allocation** in the 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654 is always greater than the sum of all asynchronous **DataWriters’ max_samples**.

### 7.5.22.2 Example

If you want to be able to store **max_samples_per_instance** for every instance, then you should set

\[ \text{max_samples} \geq \text{max_instances} \times \text{max_samples_per_instance} \]

But if you want to save memory and you do not expect that the running application will ever reach the case where it will see **max_instances** of instances, then you may use a smaller value for **max_samples** to save memory.

In any case, there is a lower limit for **max_samples**: 

\[ \text{max_samples} \geq \text{max_samples_per_instance} \]

If the 7.5.12 HISTORY QosPolicy on page 420’s **kind** is set to **KEEP_LAST**, then you should set:
max_samples_per_instance = HISTORY.depth

7.5.22.3 Properties

This QosPolicy cannot be modified after the Entity is enabled.

There are no requirements that the publishing and subscribing sides use compatible values.

7.5.22.4 Related QosPolicies

- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.21 RELIABILITY QosPolicy on page 447
- For DataReaders, max_instances must be <= max_total_instances in the 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576

7.5.22.5 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.22.6 System Resource Considerations

Larger initial_* numbers will increase the initial system memory usage. Larger max_* numbers will increase the worst-case system memory usage.

Increasing instance_hash_buckets speeds up instance-lookup time but also increases memory usage.

7.5.23 SERVICE QosPolicy (DDS Extension)

The SERVICE QosPolicy is intended for use by RTI infrastructure services. User applications should not modify its value. It includes the member in Table 7.71 DDS_ServiceQosPolicy.
7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension)

### Table 7.71 DDS_ServiceQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_ServiceQosPolicyKind</td>
<td>kind</td>
<td>Kind of service associated with the entity. Possible values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_NO_SERVICE_QOS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_PERSISTENCE_SERVICE_QOS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_QUEUING_SERVICE_QOS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_ROUTING_SERVICE_QOS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_RECORDING_SERVICE_QOS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_REPLAY_SERVICE_QOS,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_DATABASE_INTEGRATION_SERVICE_QOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_WEB_INTEGRATION_SERVICE_QOS</td>
</tr>
</tbody>
</table>

An application can determine the kind of service associated with a discovered DataWriter and DataReader by looking at the service field in the PublicationBuiltinTopicData and SubscriptionBuiltinTopicData structures (see Chapter 18 Built-In Topics on page 831).

#### 7.5.23.1 Properties

This QosPolicy cannot be modified after the Entity is enabled.

There are no requirements that the publishing and subscribing sides use compatible values.

#### 7.5.23.2 Related QosPolicies

None

#### 7.5.23.3 Applicable Entities

- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504
- 9.3 DomainParticipants on page 609

#### 7.5.23.4 System Resource Considerations

None.

#### 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension)

The TOPIC_QUERY_DISPATCH QosPolicy configures the ability of a DataWriter to publish historical samples in response to a TopicQuery (see Topic Queries (Chapter 24 on page 977)).
It contains the members listed in Table 7.72 DDS_TopicsQueryDispatchQosPolicy.

**Table 7.72 DDS_TopicsQueryDispatchQosPolicy**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>enable</td>
<td>Allows this writer to dispatch TopicQueries.</td>
</tr>
<tr>
<td>struct DDS_Duration_t</td>
<td>publication_period</td>
<td>Sets the periodic interval at which samples are published.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>samples_per_period</td>
<td>Sets the maximum number of samples to publish in each publication_period</td>
</tr>
</tbody>
</table>

This QoS policy configures the ability of a *DataWriter* to publish samples in response to a TopicQuery. It enables the ability of a *DataWriter* to publish historical samples upon reception of a TopicQuery and how often they are published.

Since a TopicQuery selects previously written samples, the *DataWriter* must have a DurabilityQosPolicy `kind` different from DDS_VOLATILE_DURABILITY_QOS. Also, the ReliabilityQosPolicy `kind` must be set to DDS_RELIABLE_RELIABILITY_QOS.

A TopicQuery may select multiple samples at once. The writer will publish them periodically, independently from newly written samples. TopicQueryDispatchQosPolicy's `publication_period` configures the frequency of that period and its `samples_per_period` configures the maximum number of samples to publish each period.

If the *DataWriter* blocks during the publication of one of these samples, it will stop and try again the next period. (See 7.3.8 Writing Data on page 309 (*FooDataWriter::write()*)) for the conditions that may cause the write operation to block.)

All the *DataWriters* that belong to a single *Publisher* and enable TopicQueries share the same event thread, but each *DataWriter* schedules separate events. To configure that thread, see the AsynchronousPublisherQosPolicy's `topic_query_publication_thread`.

If the *DataWriter* is dispatching more than one TopicQuery at the same time, the configuration of this periodic event applies to all of them. For example, if a *DataWriter* receives two TopicQueries around the same time, the period is 1 second, the number of samples per period is 10, the first TopicQuery selects five samples, and the second one selects 8, the *DataWriter* will immediately attempt to publish all five for the first TopicQuery and five for the second one. After one second, it will publish the remaining three samples.

**7.5.24.1 Properties**

This QosPolicy cannot be modified after the *Entity* is enabled.

There are no requirements that the publishing and subscribing sides use compatible values.
7.5.25 TRANSFER_MODE QosPolicy

7.5.24.2 Related QosPolicies

7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345

7.5.24.3 Applicable Entities

- 7.3 DataWriters on page 287

7.5.24.4 System Resource Considerations

None.

7.5.25 TRANSFER_MODE QosPolicy

The TRANSFER_MODE QosPolicy configures the properties of a Zero Copy DataWriter. It contains the member listed in the following table.

**Table 7.73 DDS_TransferModeQosPolicy**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>enable_data_consistency_check</td>
<td>Enables a Zero Copy DataWriter to send a special sequence number as a part of its inline Qos. This sequence number is used by a Zero Copy DataReader to check for sample consistency in the is_data_consistent() operation. For more details, see 23.1.5.1.3 Checking data consistency with Zero Copy transfer over shared memory on page 966. Default: true</td>
</tr>
</tbody>
</table>

7.5.25.1 Properties

This QosPolicy cannot be modified after the DataWriter is created.

Since it is only for DataWriters, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

7.5.25.2 Related QosPolicies

None.

7.5.25.3 Applicable Entities

7.3 DataWriters on page 287

7.5.25.4 System Resource Considerations

With enable_data_consistency_check set to true, a Zero Copy DataWriter sends an additional sequence number as part of its inline Qos with every write operation. The use of this inline QoS creates a small additional wire-payload, consuming extra bandwidth and serialization/deserialization time.
7.5.26 TRANSPORT_PRIORITY QosPolicy

The TRANSPORT_PRIORITY QosPolicy is optional and only partially supported on certain OSs and transports by RTI. However, its intention is to allow you to specify on a per-DataWriter or per-DataReader basis that the data sent by a DataWriter or DataReader is of a different priority.

DDS does not specify how a DDS implementation shall treat data of different priorities. It is often difficult or impossible for DDS implementations to treat data of higher priority differently than data of lower priority, especially when data is being sent (delivered to a physical transport) directly by the thread that called DataWriter’s write() operation. Also, many physical network transports themselves do not have an end-user controllable level of data packet priority.

In Connext DDS, for the UDPv4 built-in transport, the value set in the TRANSPORT_PRIORITY QosPolicy is used in a setsockopt call to set the TOS (type of service) bits of the IPv4 header for datagrams sent by a DataWriter or DataReader. It is platform dependent on how and whether or not the setsockopt has an effect. On some platforms such as Windows and Linux, external permissions must be given to the user application in order to set the TOS bits.

It is incorrect to assume that using the TRANSPORT_PRIORITY QosPolicy will have any effect at all on the end-to-end delivery of data between a DataWriter and DataReader. All network elements such as switches and routers must have the capability and be enabled to actually use the TOS bits to treat higher-priority packets differently. Thus the ability to use the TRANSPORT_PRIORITY QosPolicy must be designed and configured at a system level; just turning it on in an application may have no effect at all.

It includes the member in Table 7.74 DDS_TransportPriorityQosPolicy. For the default and valid range, please refer to the API Reference HTML documentation.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>value</td>
<td>Hint as to how to set the priority.</td>
</tr>
</tbody>
</table>

Connext DDS will propagate the value set on a per-DataWriter or per-DataReader basis to the transport when the DataWriter publishes data. It is up to the implementation of the transport to do something with the value, if anything.

You can set the TRANSPORT_PRIORITY QosPolicy on a Topic and use its value to initialize the TRANSPORT_PRIORITY QosPolicies of DataWriters and DataReaders. The TRANSPORT_PRIORITY QosPolicy of a Topic does not directly affect the operation of Connext DDS, see 5.1.3 Setting Topic QosPolicies on page 219.

7.5.26.1 Example

Should Connext DDS be configured with a transport that can use and will honor the concept of a prioritized message, then you would be able to create a DataWriter of a Topic whose DDS data samples,
when published, will be sent at a higher priority than other DataWriters that use the same transport.

7.5.26.2 Properties

This QosPolicy cannot be modified after the entity is created.

7.5.26.3 Related QosPolicies

This QosPolicy does not interact with any other policies.

7.5.26.4 Applicable Entities

- 5.1 Topics on page 215
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.26.5 System Resource Considerations

The use of this policy does not significantly impact the use of resources. However, if a transport is implemented to use the value set by this policy, then there may be transport-specific issues regarding the resources that the transport implementation itself uses.

7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension)

The TRANSPORT_SELECTION QosPolicy allows you to select the transports that have been installed with the DomainParticipant to be used by the DataWriter or DataReader.

An application may be simultaneously connected to many different physical transports, e.g., Ethernet, Infiniband, shared memory, VME backplane, and wireless. By default, the middleware will use up to 16 transports to deliver data from a DataWriter to a DataReader.

This QosPolicy can be used to both limit and control which of the application’s available transports may be used by a DataWriter to send data or by a DataReader to receive data.

It includes the member in Table 7.75 DDS_TransportSelectionQosPolicy. For more information, please refer to the API Reference HTML documentation.

<table>
<thead>
<tr>
<th>Table 7.75 DDS_TransportSelectionQosPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Type</strong></td>
</tr>
<tr>
<td>DDS_StringSeq</td>
</tr>
</tbody>
</table>

Connext DDS allows you to configure the transports that it uses to send and receive messages. A number of built-in transports, such as UDPv4 and shared memory, are available as well as custom ones that you
may implement and install. Each transport will be installed in the DomainParticipant with one or more aliases.

To enable a DataWriter or DataReader to use a particular transport, add the alias to the enabled_transports sequence of this QosPolicy. An empty sequence is a special case, and indicates that all transports installed in the DomainParticipant can be used by the DataWriter or DataReader.

For more information on configuring and installing transports, please see the API Reference HTML documentation (from the Modules page, select RTI DDS API Reference, Pluggable Transports).

7.5.27.1 Example

Suppose a DomainParticipant has both UDPv4 and shared memory transports installed. If you want a particular DataWriter to publish its data only over shared memory, then you should use this QosPolicy to specify that restriction.

7.5.27.2 Properties

This QosPolicy cannot be modified after the Entity is created.

It can be set differently for the DataWriter and the DataReader.

7.5.27.3 Related QosPolicies

- 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) below
- 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591
- 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665

7.5.27.4 Applicable Entities

- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

7.5.27.5 System Resource Considerations

By restricting DataWriters from sending or DataReaders from receiving over certain transports, you may decrease the load on those transports.

7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension)

The TRANSPORT_UNICAST QosPolicy allows you to specify unicast network addresses to be used by DomainParticipant, DataWriters and DataReaders for receiving messages.

Connext DDS may send data to a variety of Entities, not just DataReaders. DomainParticipants receive messages to support the discovery process discussed in Discovery (Chapter 15 on page 764). DataWriters
may receive ACK/NACK messages to support the reliable protocol discussed in Reliable Communications (Chapter 11 on page 688).

During discovery, each Entity announces to remote applications a list of (up to 16) unicast addresses to which the remote application should send data (either user-data packets or reliable protocol meta-data such as ACK/NACK and Heartbeats).

By default, the list of addresses is populated automatically with values obtained from the enabled transport plugins allowed to be used by the Entity (see the 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665 and 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459). Also, the associated ports are automatically determined (see 15.5.2 Inbound Ports for User Traffic on page 793).

Use TRANSPORT_UNICAST QosPolicy to manually set the receive address list for an Entity. You may optionally set a port to use a non-default receive port as well. Only the first 16 addresses will be used. Connect DDS will create a receive thread for every unique port number that it encounters (on a per transport basis).

The QosPolicy structure includes the members in Table 7.76 DDS_TransportUnicastQosPolicy. For more information and default values, please refer to the API Reference HTML documentation.

### Table 7.76 DDS_TransportUnicastQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TransportUnicastSettingsSeq</td>
<td>value</td>
<td>A sequence of up to 16 unicast settings that should be used by remote entities to address messages to be sent to this Entity. This is a hard limit that cannot be increased. However, this limit can be decreased by configuring the DomainParticipant property <code>dds.domain_participant.max_announced_locator_list.size</code> (whose default size is 8).</td>
</tr>
</tbody>
</table>

### Table 7.77 DDS_TransportUnicastSettings_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_StringSeq</td>
<td>transports</td>
<td>A sequence of transport aliases that specifies which transports should be used to receive unicast messages for this Entity.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>receive_port</td>
<td>The port that should be used in the addressing of unicast messages destined for this Entity. A value of 0 will cause Connect DDS to use a default port number based on domain and participant ids. See 15.5 Ports Used for Discovery on page 791.</td>
</tr>
</tbody>
</table>

A message sent to a unicast address will be received by a single node on the network (as opposed to a multicast address where a single message may be received by multiple nodes). This policy sets the unicast
addresses and ports that remote entities should use when sending messages to the Entity on which the TRANSPORT_UNICAST QosPolicy is set.

Up to 16 “return” unicast addresses may be configured for an Entity. This is a hard limit that cannot be increased. However, this limit can be decreased by configuring the DomainParticipant property dds-domain_participant.max_announced_locator_list_size. Instead of specifying addresses directly, you use the transports field of the DDS_TransportUnicastSetting_t to select the transports (using their aliases) on which remote entities should send messages destined for this Entity. The addresses of the selected transports will be the “return” addresses. See the API Reference HTML documentation about configuring transports and aliases (from the Modules page, select RTI Connext DDS API Reference, Pluggable Transports).

Note, a single transport may have more than one unicast address. For example, if a node has multiple network interface cards (NICs), then the UDPv4 transport will have an address for each NIC. When using the TRANSPORT_UNICAST QosPolicy to set the return addresses, a single value for the DDS_TransportUnicastSettingsSeq may provide more than the maximum number of return addresses that Connext DDS accepts (8 by default, changeable to 16).

Whether or not you are able to configure the network interfaces that are allowed to be used by a transport is up to the implementation of the transport. For the built-in UDPv4 transport, you may restrict an instance of the transport to use a subset of the available network interfaces. See the API Reference HTML documentation for the built-in UDPv4 transport for more information.

For a DomainParticipant, this QoS policy sets the default list of addresses used by other applications to send user data for local DataReaders.

For a reliable DataWriter, if set, the other applications will use the specified list of addresses to send reliable protocol packets (ACKS/NACKS) on the behalf of reliable DataReaders. Otherwise, if not set, the other applications will use the addresses set by the DomainParticipant.

For a DataReader, if set, then other applications will use the specified list of addresses to send user data (and reliable protocol packets for reliable DataReaders). Otherwise, if not set, the other applications will use the addresses set by the DomainParticipant.

For a DataReader, if the port number specified by this QoS is the same as a port number specified by a TRANSPORT_MULTICAST QoS, then the transport may choose to process data received both via multicast and unicast with a single thread. Whether or not a transport must use different threads to process data received via multicast or unicast for the same port number depends on the implementation of the transport.

To use this QosPolicy, you also need to specify a port number. A port number of 0 will cause Connext DDS to automatically use a default value. As explained in 15.5 Ports Used for Discovery on page 791, the default port number for unicast addresses is based on the domain and participant IDs. Should you choose to use a different port number, then for every unique port number used by Entities in your application, depending on the transport, Connext DDS may create a thread to process messages received for that port on that transport. See Connext DDS Threading Model (Chapter 21 on page 908) for more about threads.
Threads are created on a per-transport basis, so if this QosPolicy specifies multiple transports for a receive_port, then a thread may be created for each transport for that unique port. Some transports may be able to share a single thread for different ports, others can not. Different Entities can share the same port number, and thus, the same thread will process all of the data for all of the Entities sharing the same port number for a transport.

**Note:** If a DataWriter is using the 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432, the unicast addresses specified in the TRANSPORT_UNICAST QosPolicy are ignored by that DataWriter. The DataWriter will not publish DDS samples on those locators.

### 7.5.28.1 Example

You may use this QosPolicy to restrict an Entity from receiving data through a particular transport. For example, on a multi-NIC (network interface card) system, you may install different transports for different NICs. Then you can balance the network load between network cards by using different values for the TRANSPORT_UNICAST QosPolicy for different DataReaders. Thus some DataReaders will receive their data from one NIC and other DataReaders will receive their data from another.

### 7.5.28.2 Properties

This QosPolicy cannot be modified after the Entity is created.

It can be set differently for the DomainParticipant, the DataWriter and the DataReader.

### 7.5.28.3 Related QosPolicies

- 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432
- 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459
- 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591
- 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665

### 7.5.28.4 Applicable Entities

- 9.3 DomainParticipants on page 609
- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504

### 7.5.28.5 System Resource Considerations

Because this QosPolicy changes the transports on which messages are received for different Entities, the bandwidth used on the different transports may be affected.

Depending on the implementation of a transport, Connext DDS may need to create threads to receive and process data on a unique-port-number basis. Some transports can share the same thread to process data.
received for different ports; others like UDPv4 must have different threads for different ports. In addition, if the same port is used for both unicast and multicast, the transport implementation will determine whether or not the same thread can be used to process both unicast and multicast data. For UDPv4, only one thread is needed per port--independent of whether the data was received via unicast or multicast data. See 21.3 Receive Threads on page 911 for more information.

### 7.5.29 TYPESUPPORT QosPolicy (DDS Extension)

This policy can be used to modify the code generated by RTI Code Generator so that the [de]serialization routines act differently depending on the information passed in via the object pointer. This policy also determines if padding bytes are set to zero during serialization.

It includes the members in Table 7.78 DDS_TypeSupportQosPolicy.

#### Table 7.78 DDS_TypeSupportQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void *</td>
<td>plugin_data</td>
<td>Value to pass into the type plug-in's serialization/deserialization function. See Note below.</td>
</tr>
<tr>
<td>DDS_CdrPaddingKind</td>
<td>cdr_padding_kind</td>
<td>Determines whether or not the padding bytes will be set to zero during CDR serialization.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For a DomainParticipant: Configures how padding bytes are set when serializing data for the built-in topic DataWriters and DataReaders.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For DataWriters and DataReaders: Configures how padding bytes are set when serializing data for that entity.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>May be:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- ZERO_CDR_PADDING (Padding bytes will be set to zero during CDR serialization)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- NOT_SET_CDR_PADDING (Padding bytes will not be set to any value during CDR serialization)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- AUTO_CDR_PADDING (For a DomainParticipant, the default behavior is NOT_SET_CDR_PADDING. For a DataWriter or DataReader, the behavior is to inherit the value from the DomainParticipant.)</td>
</tr>
</tbody>
</table>

Note: RTI generally recommends that you treat generated source files as compiler outputs (analogous to object files) and that you do not modify them. RTI cannot support user changes to generated source files. Furthermore, such changes would make upgrading to newer versions of Connext DDS more difficult, as this generated code is considered to be a part of the middleware implementation and consequently does change from version to version. The plugin_data field in this QoS policy should be considered a back door, only to be used after careful design consideration, testing, and consultation with your RTI representative.

#### 7.5.29.1 Properties

This QoS policy may be modified after the DataWriter or DataReader is enabled.

It can be set differently for the DataWriter and DataReader.
7.5.29.2 Related QoS Policies

None.

7.5.29.3 Applicable Entities

- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504
- 9.3 DomainParticipants on page 609

7.5.29.4 System Resource Considerations

None.

7.5.30 USER_DATA QosPolicy

This QosPolicy provides an area where your application can store additional information related to a DomainParticipant, DataWriter, or DataReader. This information is passed between applications during discovery (see Discovery (Chapter 15 on page 764)) using built-in-topics (see Built-In Topics (Chapter 18 on page 831)). How this information is used will be up to user code. Connext DDS does not do anything with the information stored as USER_DATA except to pass it to other applications.

Use cases are usually for application-to-application identification, authentication, authorization, and encryption purposes. For example, applications can use Group or User Data to send security certificates to each other for RSA-type security.

The value of the USER_DATA QosPolicy is sent to remote applications when they are first discovered, as well as when the DomainParticipant, DataWriter or DataReader’s set_qos() methods are called after changing the value of the USER_DATA. User code can set listeners on the built-in DataReaders of the built-in Topics used by Connext DDS to propagate discovery information. Methods in the built-in topic listeners will be called whenever new DomainParticipants, DataReaders, and DataWriters are found. Within the user callback, you will have access to the USER_DATA that was set for the associated Entity.

Currently, USER_DATA of the associated Entity is only propagated with the information that declares a DomainParticipant, DataWriter or DataReader. Thus, you will need to access the value of USER_DATA through DDS_ParticipantBuiltinTopicData, DDS_PublicationBuiltinTopicData or DDS_SubscriptionBuiltinTopicData (see Built-In Topics (Chapter 18 on page 831)).

The structure for the USER_DATA QosPolicy includes just one field, as seen in Table 7.79 DDS_UserDataQosPolicy. The field is a sequence of octets that translates to a contiguous buffer of bytes whose contents and length is set by the user. The maximum size for the data are set in the 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654.
Table 7.79 DDS_UserDataQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_OctetSeq</td>
<td>value</td>
<td>Default: empty</td>
</tr>
</tbody>
</table>

This policy is similar to the 7.4.4 GROUP_DATA QosPolicy on page 353 and 5.2.1 TOPIC_DATA QosPolicy on page 224 that apply to other types of Entities.

7.5.30.1 Example

One possible use of USER_DATA is to pass some credential or certificate that your subscriber application can use to accept or reject communication with the DataWriters (or vice versa, where the publisher application can validate the permission of DataReaders to receive its data). Using the same method, an application (DomainParticipant) can accept or reject all connections from another application. The value of the USER_DATA of the DomainParticipant is propagated in the ‘user_data’ field of the DDS_ParticipantBuiltinTopicData that is sent with the declaration of each DomainParticipant. Similarly, the value of the USER_DATA of the DataWriter is propagated in the ‘user_data’ field of the DDS_PublicationBuiltinTopicData that is sent with the declaration of each DataWriter, and the value of the USER_DATA of the DataReader is propagated in the ‘user_data’ field of the DDS_SubscriptionBuiltinTopicData that is sent with the declaration of each DataReader.

When Connext DDS discovers a DomainParticipant/DataWriter/DataReader, the application can be notified of the discovery of the new entity and retrieve information about the Entity’s QoS by reading the DCPSParticipant, DCPSPublication or DCPSSubscription built-in topics (see Built-In Topics (Chapter 18 on page 831)). The user application can then examine the USER_DATA field in the built-in Topic and decide whether or not the remote Entity should be allowed to communicate with the local Entity. If communication is not allowed, the application can use the DomainParticipant’s ignore_participant(), ignore_publication() or ignore_subscription() operation to reject the newly discovered remote entity as one with which the application allows Connext DDS to communicate. See 18.2 Built-in DataReaders on page 832 for an example of how to do this.

7.5.30.2 Properties

This QosPolicy can be modified at any time. A change in the QosPolicy will cause Connext DDS to send packets containing the new USER_DATA to all of the other applications in the DDS domain.

It can be set differently on the publishing and subscribing sides.

7.5.30.3 Related QosPolicies

- 5.2.1 TOPIC_DATA QosPolicy on page 224
- 7.4.4 GROUP_DATA QosPolicy on page 353
7.5.31 WRITER_DATA_LIFECYCLE QoS Policy

- 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654

7.5.30.4 Applicable Entities

- 7.3 DataWriters on page 287
- 8.3 DataReaders on page 504
- 9.3 DomainParticipants on page 609

7.5.30.5 System Resource Considerations

The maximum size of the USER_DATA is set in the participant_user_data_max_length, writer_user_data_max_length, and reader_user_data_max_length fields of the 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654. Because Connext DDS will allocated memory based on this value, you should only increase this value if you need to. If your system does not use USER_DATA, then you can set this value to 0 to save memory. Setting the value of the USER_DATA QosPolicy to hold data longer than the value set in the [participant,writer,reader]_user_data_max_length field will result in failure and an INCONSISTENT_QOS_POLICY return code.

However, should you decide to change the maximum size of USER_DATA, you must make certain that all applications in the DDS domain have changed the value of [participant,writer,reader]_user_data_max_length to be the same. If two applications have different limits on the size of USER_DATA, and one application sets the USER_DATA QosPolicy to hold data that is greater than the maximum size set by another application, then the DataWriters and DataReaders between the two applications will not connect. The DomainParticipants may also reject connections from each other entirely. This is also true for the GROUP_DATA (7.4.4 GROUP_DATA QosPolicy on page 353) and TOPIC_DATA (5.2.1 TOPIC_DATA QosPolicy on page 224) QosPolicies.

7.5.31 WRITER_DATA_LIFECYCLE QoS Policy

This QoS policy controls how a DataWriter handles the lifecycle of the instances (keys) that the DataWriter is registered to manage. This QoS policy includes the members in Table 7.80 DDS_Writer-DataLifecycleQosPolicy.
Table 7.80 DDS_WriterDataLifecycleQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| DDS_Boolean   | autodispose_unregistered_instances | Controls what happens when the DataWriter unregisters an instance by means of the unregister operations. This setting has no impact on the DataWriter deletion operation. When a DataWriter is deleted, and it was the last known DataWriter for any of the instances that it was writing, the instance will automatically be transitioned to NOT_ALIVE_NO_WRITERS by all matching DataReaders. No unregister messages are sent in this scenario, and therefore no dispose messages are sent, either, regardless of the value of this setting. RTI_TRUE: The DataWriter first disposes of the instance each time it unregisters from the instance. This behavior is identical to explicitly calling one of the dispose operations on the instance prior to calling the unregister operation.
   
   Note: It is recommended that you keep this QoS setting at FALSE. See 7.5.31.2 Autodisposing Unregistered Instances on the next page.

RTI_FALSE (default): The DataWriter does not dispose of the instance each time it is unregistered. The application can still call one of the dispose operations prior to unregistering the instance and dispose of the instance that way. When a DataWriter is deleted, no extra messages are sent. DataReaders will automatically unregister this DataWriter from all instances when they recognize that the DataWriter has been deleted.  

| struct DDS_Duration_t | autopurge_unregistered_instances_delay | Determines how long the DataWriter will maintain information regarding an instance that has been unregistered. By default, the DataWriter resources associated with an instance (e.g., the space needed to remember the Instance Key or KeyHash) are released lazily. This means the resources are only reclaimed when the space is needed for another instance because max_instances (7.5.22 RESOURCE_LIMITS QosPolicy on page 451) is exceeded. This behavior can be changed by setting autopurge_unregistered_instances_delay to a value other than INFINITE. After this time elapses, the DataWriter will purge all internal information regarding the instance, including historical DDS samples even if max_instances has not been reached.

The purging of unregistered instances can be done based on the source timestamp of the unregister sample or the time when the unregister sample was added to the DataWriter queue, by setting the following property to 1 or 0 respectively (default: 0): dds.data_writer.history.source_timestamp_basedautopurge_instances_delay. The source timestamp can differ from the time that the sample was added to the queue if a timestamp was provided along with the sample when it was written (using the write_with_timestamp() or write_with_params() operations). This is the case, for example, in RTI Routing Service when samples are routed with the original publisher information.

For durable writer history, autopurge_unregistered_instances_delay supports only the INFINITE value. Default: INFINITE (except for builtin DataWriters, in which case 0)  

| struct DDS_Duration_t | autopurge_disposed_instances_delay | Determines the maximum duration for which the DataWriter will maintain information regarding an instance once it has disposed of the instance. By default, disposing of an instance does not make it eligible to be purged. By setting autopurge_disposed_instances_delay to a value other than DDS_DURATION_INFINITE, the DataWriter will reclaim the resources associated with an instance (including historical samples) once the time has elapsed and all matching DataReaders have acknowledged all the samples for this instance, including the dispose sample.

The purging of the disposed instances can be done based on the dispose sample source timestamp or the time when the dispose sample was added to the DataWriter queue, by setting the following property to 1 or 0 respectively (default: 0): dds.data_writer.history.source_timestamp_basedautopurge_instances_delay. The source timestamp can differ from the time that the sample was added to the queue if a timestamp was provided along with the sample when it was written (using the write_with_timestamp() or write_with_params() operations). This is the case, for example, in Routing Service when samples are routed with the original publisher information.

autopurge_disposed_instances_delay is supported with durable DataWriter queues only for 0 and INFINITE values (finite values are not supported). Default: INFINITE |
7.5.31.1 Unregistering vs. Disposing

- Disposing an instance conveys an explicit state about an instance: for example, disposing a flight because it has landed. You can decide what dispose means for your system. See 7.3.14.3 Disposing Instances on page 328.

- Unregistering an instance can be thought of as a DataWriter unregistering itself from the instance, indicating that the DataWriter has no more information/data on the instance. An example is when radar is no longer tracking a flight. In this example, the flight is still a valid, alive instance in the system, with the same location and trajectory, but this specific radar is simply no longer tracking it. Unregistering tells Connext DDS that the DataWriter does not intend to modify that instance anymore, allowing Connext DDS to recover any resources it allocated for the instance. See 7.3.14.4 Unregistering Instances on page 328.

7.5.31.2 Autodisposing Unregistered Instances

The autodispose_unregistered_instances QoS setting determines whether explicit calls to an unregister operation also automatically first dispose the instance that it is being unregistered from.

It is recommended to keep the default setting of FALSE for autodispose_unregistered_instances and manage all instance state transitions through explicit calls to dispose() and unregister_instance() in your application. The reasons for this recommendation are as follows:

- In many cases where the ownership of a Topic is EXCLUSIVE (see the 7.5.17 OWNERSHIP QosPolicy on page 434), DataWriters may want to relinquish ownership of a particular instance of the Topic to allow other DataWriters to send updates for the value of that instance. In this case, you may want a DataWriter to just unregister an instance—without disposing it (since there are other writers). Unregistering an instance implies that the DataWriter no longer owns that instance, but it is a stronger statement to say that instance no longer exists.

- User applications may be coded to trigger on the disposal of instances, thus the ability to unregister without disposing may be useful to properly maintain the semantic of disposal.

7.5.31.3 Properties

The WRITER_DATA_LIFECYCLE QoS Policy does not apply to DataReaders, so there is no requirement that the publishing and subscribing sides use compatible values.

This QoS policy may be modified after the DataWriter is enabled.

7.5.31.4 Related QoS Policies

- None.
7.6 FlowControllers (DDS Extension)

This section does not apply when using the separate add-on product, Ada Language Support, which does not support FlowControllers.

A FlowController is the object responsible for shaping the network traffic by determining when attached asynchronous DataWriters are allowed to write data. To configure a DataWriter to be asynchronous, see 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444.

You can use one of the built-in FlowControllers (and optionally modify their properties), create a custom FlowController by using the DomainParticipant's create_flowcontroller() operation (see 7.6.6 Creating and Deleting FlowControllers on page 481), or create a custom FlowController by using the DomainParticipant's 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439; see 7.6.5 Creating and Configuring Custom FlowControllers with Property QoS on page 479.

To use a FlowController, you provide its name in the DataWriter's 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444.

- **DDS_DEFAULT_FLOW_CONTROLLER_NAME**
  
  By default, flow control is disabled. That is, the built-in DDS_DEFAULT_FLOW_CONTROLLER_NAME flow controller does not apply any flow control. Instead, it allows data to be sent asynchronously as soon as it is written by the DataWriter.

- **DDS_FIXED_RATE_FLOW_CONTROLLER_NAME**
  
  The FIXED_RATE flow controller shapes the network traffic by allowing data to be sent only once every second. Any accumulated DDS samples destined for the same destination are coalesced into as few network packets as possible.

- **DDS_ON_DEMAND_FLOW_CONTROLLER_NAME**
  
  The ON_DEMAND flow controller allows data to be sent only when you call the FlowController’s trigger_flow() operation. With each trigger, all accumulated data since the previous trigger is sent (across all Publishers or DataWriters). In other words, the network traffic shape is fully controlled by the user. Any accumulated DDS samples destined for the same destination are coalesced into as
few network packets as possible.

This external trigger source is ideal for users who want to implement some form of closed-loop flow control or who want to only put data on the wire every so many DDS samples (e.g., with the number of DDS samples based on NDDS_Transport_Property_t's `gather_send_buffer_count_max`).

The default property settings for the built-in FlowControllers are described in the API Reference HTML documentation.

DDS samples written by an asynchronous `DataWriter` are not sent in the context of the `write()` call. Instead, Connext DDS puts the DDS samples in a queue for future processing and they are sent in the asynchronous publishing thread. (See 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345.) The FlowController associated with each asynchronous `DataWriter` determines when the DDS samples are actually sent.

Each FlowController maintains a separate FIFO queue for each unique destination (remote application). DDS samples written by asynchronous `DataWriters` associated with the FlowController are placed in the queues that correspond to the intended destinations of the DDS sample.

When tokens become available, a FlowController must decide which queue(s) to grant tokens first. This is determined by the FlowController's `scheduling_policy` property (see Table 7.81 DDS_FlowControllerProperty_t). Once a queue has been granted tokens, it is serviced by the asynchronous publishing thread. The queued up DDS samples will be coalesced and sent to the corresponding destination. The number of DDS samples sent depends on the data size and the number of tokens granted.

Table 7.81 DDS_FlowControllerProperty_t lists the properties for a FlowController.

Table 7.81 DDS_FlowControllerProperty_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_FlowControllerSchedulingPolicy</td>
<td>scheduling_policy</td>
<td>Round robin, earliest deadline first, or highest priority first. See 7.6.1 Flow Controller Scheduling Policies on the next page.</td>
</tr>
<tr>
<td>DDS_FlowControllerTokenBucketProperty_t</td>
<td>token_bucket</td>
<td>See 7.6.3 Token Bucket Properties on page 474.</td>
</tr>
</tbody>
</table>

Table 7.82 FlowController Operations lists the operations available for a FlowController.
Table 7.82 FlowController Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_property</td>
<td>Get and Set the FlowController properties.</td>
<td>7.6.8 Getting/Setting Properties for a Specific FlowController on page 482</td>
</tr>
<tr>
<td>set_property</td>
<td></td>
<td></td>
</tr>
<tr>
<td>trigger_flow</td>
<td>Provides an external trigger to the FlowController.</td>
<td>7.6.9 Adding an External Trigger on page 483</td>
</tr>
<tr>
<td>get_name</td>
<td>Returns the name of the FlowController.</td>
<td></td>
</tr>
<tr>
<td>get_participant</td>
<td>Returns the DomainParticipant to which the FlowController belongs.</td>
<td>7.6.10 Other FlowController Operations on page 483</td>
</tr>
</tbody>
</table>

7.6.1 Flow Controller Scheduling Policies

- **Round Robin**
  
  (DDS_RR_FLOW_CONTROLLER_SCHED_POLICY) Perform flow control in a round-robin (RR) fashion.

  Whenever tokens become available, the FlowController distributes the tokens uniformly across all of its (non-empty) destination queues. No destinations are prioritized. Instead, all destinations are treated equally and are serviced in a round-robin fashion.

- **Earliest Deadline First**
  
  (DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY) Perform flow control in an earliest-deadline-first (EDF) fashion.

  A DDS sample's deadline is determined by the time it was written plus the latency budget of the DataWriter at the time of the write call (as specified in the DDS_LatencyBudgetQosPolicy). The relative priority of a flow controller's destination queue is determined by the earliest deadline across all DDS samples it contains.

  When tokens become available, the FlowController distributes tokens to the destination queues in order of their priority. In other words, the queue containing the DDS sample with the earliest deadline is serviced first. The number of tokens granted equals the number of tokens required to send the first DDS sample in the queue. Note that the priority of a queue may change as DDS samples are sent (i.e., removed from the queue). If a DDS sample must be sent to multiple destinations or two DDS samples have an equal deadline value, the corresponding destination queues are serviced in a round-robin fashion.

  With the default duration of 0 in the LatencyBudgetQosPolicy, using an EDF_FLOW_CONTROLLER_SCHED_POLICY FlowController preserves the order in which you call write() across the DataWriters associated with the FlowController.
7.6.2 Managing Fast DataWriters When Using a FlowController

Since the LatencyBudgetQosPolicy is mutable, a DDS sample written second may contain an earlier deadline than the DDS sample written first if the DDS_LatencyBudgetQosPolicy’s **duration** is sufficiently decreased in between writing the two DDS samples. In that case, if the first DDS sample is not yet written (still in queue waiting for its turn), it inherits the priority corresponding to the (earlier) deadline from the second DDS sample.

In other words, the priority of a destination queue is always determined by the earliest deadline among all DDS samples contained in the queue. This priority inheritance approach is required in order to both honor the updated **duration** and to adhere to the DataWriter in-order data delivery guarantee.

- **Highest Priority First**

(DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY) Perform flow control in an highest-priority-first (HPF) fashion.

**Note:** Prioritized DDS samples are not supported when using the Ada API. Therefore, the Highest Priority First scheduling policy is not supported when using this API.

The next destination queue to service is determined by the publication priority of the DataWriter, the channel of a multi-channel DataWriter, or individual DDS sample.

The relative priority of a flow controller's destination queue is determined by the highest publication priority of all the DDS samples it contains.

When tokens become available, the FlowController distributes tokens to the destination queues in order of their publication priority. The queue containing the DDS sample with the highest publication priority is serviced first. The number of tokens granted equals the number of tokens required to send the first DDS sample in the queue. Note that a queue’s priority may change as DDS samples are sent (i.e., as they are removed from the queue). If a DDS sample must be sent to multiple destinations or two DDS samples have the same publication priority, the corresponding destination queues are serviced in a round-robin fashion.

This priority inheritance approach is required to both honor the designated publication priority and adhere to the DataWriter’s in-order data delivery guarantee.

See also: 7.6.4 Prioritized DDS Samples on page 476.

7.6.2 Managing Fast DataWriters When Using a FlowController

If a DataWriter is writing DDS samples faster than its attached FlowController can throttle, Connext DDS may drop DDS samples on the writer’s side. This happens because the DDS samples may be removed from the queue before the asynchronous publisher’s thread has a chance to send them. To work around this problem, either:
- Use reliable communication to block the `write()` call and thereby throttle your application.
- Do not allow the queue to fill up in the first place.

The queue should be sized large enough to handle expected write bursts, so that no DDS samples are dropped. Then in steady state, the FlowController will smooth out these bursts and the queue will ideally have only one entry.

### 7.6.3 Token Bucket Properties

FlowControllers use a token-bucket approach for open-loop network flow control. The flow control characteristics are determined by the token bucket properties. The properties are listed in Table 7.83 `DDS_FlowControllerTokenBucketProperty_t`; see the API Reference HTML documentation for their defaults and valid ranges.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_tokens</td>
<td>Maximum number of tokens than can accumulate in the token bucket. See 7.6.3.1 max_tokens on the next page.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>tokens_added_per_period</td>
<td>The number of tokens added to the token bucket per specified period. See 7.6.3.2 tokens_added_per_period on the next page.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>tokens_leaked_per_period</td>
<td>The number of tokens removed from the token bucket per specified period. See 7.6.3.3 tokens_leaked_per_period on the next page.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>period</td>
<td>Period for adding tokens to and removing tokens from the bucket. See 7.6.3.4 period on the next page.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>bytes_per_token</td>
<td>Maximum number of bytes allowed to send for each token available. See 7.6.3.5 bytes_per_token on the next page.</td>
</tr>
</tbody>
</table>

Asynchronously published DDS samples are queued up and transmitted based on the token bucket flow control scheme. The token bucket contains tokens, each of which represents a number of bytes. DDS samples can be sent only when there are sufficient tokens in the bucket. As DDS samples are sent, tokens are consumed. The number of tokens consumed is proportional to the size of the data being sent. Tokens are replenished on a periodic basis.

The rate at which tokens become available and other token bucket properties determine the network traffic flow.

Note that if the same DDS sample must be sent to multiple destinations, separate tokens are required for each destination. Only when multiple DDS samples are destined to the same destination will they be coalesced and sent using the same token(s). In other words, each token can only contribute to a single network packet.
7.6.3 Token Bucket Properties

7.6.3.1 max_tokens

The maximum number of tokens in the bucket will never exceed this value. Any excess tokens are discarded. This property value, combined with bytes_per_token, determines the maximum allowable data burst.

Use DDS_LENGTH_UNLIMITED to allow accumulation of an unlimited amount of tokens (and therefore potentially an unlimited burst size).

7.6.3.2 tokens_added_per_period

A FlowController transmits data only when tokens are available. Tokens are periodically replenished. This field determines the number of tokens added to the token bucket with each periodic replenishment.

Available tokens are distributed to associated DataWriters based on the scheduling_policy. Use DDS_LENGTH_UNLIMITED to add the maximum number of tokens allowed by max_tokens.

7.6.3.3 tokens_leaked_per_period

When tokens are replenished and there are sufficient tokens to send all DDS samples in the queue, this property determines whether any or all of the leftover tokens remain in the bucket.

Use DDS_LENGTH_UNLIMITED to remove all excess tokens from the token bucket once all DDS samples have been sent. In other words, no token accumulation is allowed. When new DDS samples are written after tokens were purged, the earliest point in time at which they can be sent is at the next periodic replenishment.

7.6.3.4 period

This field determines the period by which tokens are added or removed from the token bucket.

The special value DDS_DURATION_INFINITE can be used to create an on-demand FlowController, for which tokens are no longer replenished periodically. Instead, tokens must be added explicitly by calling the FlowController’s trigger_flow() operation. This external trigger adds tokens_added_per_period tokens each time it is called (subject to the other property settings).

Once period is set to DDS_DURATION_INFINITE, it can no longer be reverted to a finite period.

7.6.3.5 bytes_per_token

This field determines the number of bytes that can actually be transmitted based on the number of tokens.

Tokens are always consumed in whole by each DataWriter. That is, in cases where bytes_per_token is greater than the DDS sample size, multiple DDS samples may be sent to the same destination using a single token (regardless of the scheduling_policy).
Where fragmentation is required, the fragment size will be either (a) `bytes_per_token` or (b) the minimum of the `message_size_max` transport configuration across all transports installed with the `DataWriter`, whichever is less. See information about `message_size_max` in the desired transport, such as Table 16.2 Properties for the Builtin UDPv4 Transport.

Use `DDS_LENGTH_UNLIMITED` to indicate that an unlimited number of bytes can be transmitted per token. In other words, a single token allows the recipient `DataWriter` to transmit all its queued DDS samples to a single destination. A separate token is required to send to each additional destination.

### 7.6.4 Prioritized DDS Samples

**Note:** This feature is not supported when using the Ada API.

The *Prioritized DDS Samples* feature allows you to prioritize traffic that is in competition for transmission resources. The granularity of this prioritization may be by `DataWriter`, by instance, or by individual DDS sample.

*Prioritized DDS Samples* can improve latency in the following cases:

- **Low-Availability Links**
  
  With low-availability communication, unsent DDS samples may accumulate while the link is unavailable. When the link is restored, a large number of DDS samples may be waiting for transmission. High priority DDS samples will be sent first.

- **Low-Bandwidth Links**
  
  With low-bandwidth communication, a temporary backlog may occur or the link may become congested with large DDS samples. High-priority DDS samples will be sent at the first available gap, between the fragments of a large low-priority DDS sample.

- **Prioritized Topics**
  
  With limited bandwidth communication, some topics may be deemed to be of higher priority than others on an ongoing basis, and DDS samples written to some topics should be given precedence over others on transmission.

- **High Priority Events**
  
  Due to external rules or content analysis (e.g., perimeter violation or identification as a threat), the priority of DDS samples is dynamically determined, and the priority assigned a given DDS sample will reflect the urgency of its delivery.
To configure a DataWriter to use prioritized DDS samples:

- Create a FlowController with the `scheduling_policy` property set to `DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY`.
- Create a DataWriter with the 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444 kind set to `ASYNCHRONOUS` and `flow_controller_name` set to the name of the FlowController.

A single FlowController may perform traffic shaping for multiple DataWriters and multiple DataWriter channels. The FlowController’s configuration determines how often publication resources are scheduled, how much data may be sent per period, and other transmission characteristics that determine the ultimate performance of prioritized DDS samples.

When working with prioritized DDS samples, you should use these operations, which allow you to specify priority:

- `write_w_params()` (see 7.3.8 Writing Data on page 309)
- `unregister_instance_w_params()` (see 7.3.14.4 Unregistering Instances on page 328)
- `dispose_w_params()` (see 7.3.14.3 Disposing Instances on page 328)

If you use `write()`, `unregister()`, or `dispose()` instead of the `_w_params()` versions, the affected DDS sample is assigned priority 0 (undefined priority). If you are using a multi-channel DataWriter with a priority filter, and you have no channel for priority 0, the DDS sample will be discarded.

### 7.6.4.1 Designating Priorities

For DataWriters and DataWriter channels, valid publication priority values are:

- DDS_PUBLICATION_PRIORITY_UNDEFINED
- DDS_PUBLICATION_PRIORITY_AUTOMATIC
- Positive integers excluding zero

For individual DDS samples, valid publication priority values are 0 and positive integers.

There are three ways to set the publication priority of a DataWriter or DataWriter channel:

1. For a DataWriter, publication priority is set in the `priority` field of its 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444. For a multi-channel DataWriter (see 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432), this value will be the default publication priority for any member channel that has not been assigned a specific value.
2. For a channel of a Multi-channel DataWriter, publication priority can be set in the DataWriter’s 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432 in channels[].priority.

3. If a DataWriter or a channel of a Multi-channel DataWriter is configured for publication priority inheritance (DDS_PUBLICATION_PRIORITY_AUTOMATIC), its publication priority is the highest priority among all the DDS samples currently in the publication queue. When using publication priority inheritance, the publication priorities of individual DDS samples are set by calling the write_w_params() operation, which takes a priority parameter.

The effective publication priority is determined from the interaction of the DataWriter, channel, and DDS sample publication priorities, as shown in Table 7.84 Effective Publication Priority of Samples.

Table 7.84 Effective Publication Priority of Samples

<table>
<thead>
<tr>
<th>Writer Priority</th>
<th>Channel Priority</th>
<th>DDS Sample Priority</th>
<th>Effective Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>Undefined</td>
<td>Undefined</td>
<td>Don’t care</td>
<td>Lowest Priority</td>
</tr>
<tr>
<td>Don’t care</td>
<td>AUTOMATIC</td>
<td>Designated positive integer &gt; 0</td>
<td>DDS Sample Priority&lt;sup&gt;1&lt;/sup&gt;</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>Designated positive integer &gt; 0</td>
<td>Don’t care</td>
<td>Channel Priority</td>
</tr>
<tr>
<td>Don’t care</td>
<td>Don’t care</td>
<td>Designated positive integer &gt; 0</td>
<td>Don’t care</td>
</tr>
<tr>
<td>Designated positive integer &gt; 0</td>
<td>Don’t care</td>
<td>Designated positive integer &gt; 0</td>
<td>Writer Priority</td>
</tr>
</tbody>
</table>

7.6.4.2 Priority-Based Filtering

The configuration methods explained above are sufficient to create multiple DataWriters, each with its own assigned priority, all using the same FlowController configured for publication priority-based scheduling. Such a configuration is sufficient to assign different priorities to individual topics, but it does not allow different publication priorities to be assigned to published data within a Topic.

To assign different priorities to data within a DataWriter, you will need to use a Multi-channel DataWriter and configure the channels with different priorities. Configuring the publication priorities of DataWriter channels is explained above. To associate different priorities of data with different publication channels, configure the channel[].filter_expression in the DataWriter’s 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432. The filtering criteria that is available for evaluation by each channel is determined by the filter type, which is configured with the DataWriter’s filter_name (also in the 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432).

<sup>1</sup>Highest sample priority among all DDS samples currently in the publication queue.

<sup>2</sup>Highest sample priority among all DDS samples currently in the publication queue.
For example, using the built-in SQL-based content filter allows channel membership to be determined based on the content of each DDS sample.

If you do not want to embed priority criteria within each DDS sample, you can use a built-in filter named DDS_PRIFILTER_NAME that uses the publication priority that is provided when you call `write_w_params()` (see 7.3.8 Writing Data on page 309). The filter's expression syntax is:

@priority OP VAL

where OP can be <, <=, >, >=, =, or <> (standard relational operators), and VAL is a positive integer.

The filter supports multiple expressions, combined with the conjunctions AND and OR. You can use parentheses to disambiguate combinations of AND and OR in the same expression. For example:

@priority = 2 OR (@priority > 6 AND @priority < 10)

### 7.6.5 Creating and Configuring Custom FlowControllers with Property QoS

You can create and configure FlowControllers using the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439. The properties must have a prefix of “dds.flow_controller.token_bucket”, followed by the name of the FlowController being created or configured. For example, if you want to create/configure a FlowController named MyFC, all the properties for MyFC should have the prefix “dds.flow_controller.token_bucket.MyFC”.

Table 7.85 FlowController Properties lists the properties that can be set for FlowControllers in the DomainParticipant's 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439. A FlowController with the name "dds.flow_controller.token_bucket.<your flow controller name>" will be implicitly created when at least one property using that prefix is specified. Then, to link a DataWriter to your FlowController, use "dds.flow_controller.token_bucket.<your flow controller name>" in the DataWriter's publish_mode.flow_controller_name.

### Table 7.85 FlowController Properties

<table>
<thead>
<tr>
<th>Property Name prefix with 'dds.flow_controller.token_bucket. &lt;your flow controller name&gt;'</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheduling_policy</td>
<td>Specifies the scheduling policy to be used. (See 7.6.1 Flow Controller Scheduling Policies on page 472) May be: DDS_RR_FLOW_CONTROLLER_SCHED_POLICY DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY DDS_HPF_FLOW_CONTROLLER_SCHED_POLICY</td>
</tr>
<tr>
<td>token_bucket.max_tokens</td>
<td>Maximum number of tokens than can accumulate in the token bucket. Use -1 for unlimited.</td>
</tr>
</tbody>
</table>
### Table 7.85 FlowController Properties

<table>
<thead>
<tr>
<th>Property Name prefix with <code>dds.flow_controller.token_bucket. &lt;your flow controller name&gt;</code></th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>token_bucket.tokens_added_per_period</code></td>
<td>Number of tokens added to the token bucket per specified period. Use <code>-1</code> for unlimited.</td>
</tr>
<tr>
<td><code>token_bucket.tokens_leaked_per_period</code></td>
<td>Number of tokens removed from the token bucket per specified period. Use <code>-1</code> for unlimited.</td>
</tr>
<tr>
<td><code>token_bucket.period.sec</code></td>
<td>Period for adding tokens to and removing tokens from the bucket in seconds.</td>
</tr>
<tr>
<td><code>token_bucket.period.nanosec</code></td>
<td>Period for adding tokens to and removing tokens from the bucket in nanoseconds.</td>
</tr>
<tr>
<td><code>token_bucket.bytes_per_token</code></td>
<td>Maximum number of bytes allowed to send for each token available.</td>
</tr>
</tbody>
</table>

#### 7.6.5.1 Example

The following example shows how to set FlowController properties.

```xml
<domain_participant_qos>
  <property>
    <value>
      <element>
        <name>
          dds.flow_controller.token_bucket.MyFlowController.scheduling_policy
        </name>
        <value>DDS_RR_FLOW_CONTROLLER_SCHED_POLICY</value>
      </element>
      <element>
        <name>
          dds.flow_controller.token_bucket.MyFlowController.token_bucket.period.sec
        </name>
        <value>100</value>
      </element>
      <element>
        <name>
          dds.flow_controller.token_bucket.MyFlowController.token_bucket.period.nanosec
        </name>
        <value>0</value>
      </element>
      <element>
        <name>
          dds.flow_controller.token_bucket.MyFlowController.token_bucket.tokens_added_per_period
        </name>
        <value>2</value>
      </element>
      <element>
        <name>
          dds.flow_controller.token_bucket.MyFlowController.token_bucket.tokens_leaked_per_period
        </name>
        <value>480</value>
      </element>
    </value>
  </property>
</domain_participant_qos>
```
7.6.6 Creating and Deleting FlowControllers

(Note: in the Modern C++ API FlowControllers have reference semantics, see Creating and Deleting Entities)

If you do not want to use one of the three built-in FlowControllers described in 7.6 FlowControllers (DDS Extension) on page 470, you can create your own with the DomainParticipant’s create_flowcontroller() operation:

```cpp
static DDSFlowController* create_flowcontroller
(const char * name,
 const DDS_FlowControllerProperty_t & property)
```

To associate a FlowController with a DataWriter, you set the FlowController’s name in the 7.5.20 PUBLISH_MODE_qosPolicy (DDS Extension) on page 444 (flow_controller_name).

A single FlowController may service multiple DataWriters, even if they belong to a different Publisher. The FlowController’s property structure determines how the FlowController shapes the network traffic.

- **name**: Name of the FlowController to create. A DataWriter is associated with a DDSFlowController by name. Limited to 255 characters.
- **property**: Properties to be used for creating the FlowController. The special value DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT can be used to indicate that the FlowController should be created with the default DDS_FlowControllerProperty_t set in the DomainParticipant.

**Note**: If you use DDS_FLOW_CONTROLLER_PROPERTY_DEFAULT, it is not safe to create the FlowController while another thread may be simultaneously calling set_default_flowcontroller_property() or looking for that FlowController with lookup_flowcontroller().

To delete an existing FlowController, use the DomainParticipant’s delete_flowcontroller() operation:
The FlowController must belong to a DomainParticipant and not have any attached DataWriters or the delete call will return an error (PRECONDITION_NOT_MET).

### 7.6.7 Getting/Setting Default FlowController Properties

To get the default DDS_FlowControllerProperty_t values, use this operation on the DomainParticipant:

```c
DDS_ReturnCode_t get_default_flowcontroller_property
  (DDS_FlowControllerProperty_t & property)
```

The retrieved property will match the set of values specified on the last successful call to the DomainParticipant’s `set_default_flowcontroller_property()`, or if the call was never made, the default values listed in DDS_FlowControllerProperty_t.

To change the default DDS_FlowControllerProperty_t values used when a new FlowController is created, use this operation on the DomainParticipant:

```c
DDS_ReturnCode_t set_default_flowcontroller_property
  (const DDS_FlowControllerProperty_t & property)
```

The special value DDS_FLOW_CONTROLLERPROPERTY_DEFAULT may be passed for the property to indicate that the default property should be reset to the default values the factory would use if `set_default_flowcontroller_property()` had never been called.

**Note:** It is not safe to set the default FlowController properties while another thread may be simultaneously calling `get_default_flowcontroller_property()`, `set_default_flowcontroller_property()`, or `create_flowcontroller()` with DDS_FLOW_CONTROLLERPROPERTY_DEFAULT as the qos parameter. It is also not safe to get the default FlowController properties while another thread may be simultaneously calling `get_default_flowcontroller_property()`.

### 7.6.8 Getting/Setting Properties for a Specific FlowController

To get the properties of a FlowController, use the FlowController’s `get_property()` operation:

```c
DDS_ReturnCode_t DDSFlowController::get_property
  (struct DDS_FlowControllerProperty_t & property)
```

To change the properties of a FlowController, use the FlowController’s `set_property()` operation:

```c
DDS_ReturnCode_t DDSFlowController::set_property
  (const struct DDS_FlowControllerProperty_t & property)
```

Once a FlowController has been instantiated, only its `token_bucket` property can be changed. The `scheduling_policy` is immutable. A new `token.period` only takes effect at the next scheduled token distribution time (as determined by its previous value).

The special value DDS_FLOW_CONTROLLERPROPERTY_DEFAULT can be used to match the current default properties set in the DomainParticipant.
7.6.9 Adding an External Trigger

Typically, a FlowController uses an internal trigger to periodically replenish its tokens. The period by which this trigger is called is determined by the **period** property setting.

The **trigger_flow()** function provides an additional, external trigger to the FlowController. This trigger adds **tokens_added_per_period** tokens each time it is called (subject to the other property settings of the FlowController).

```cpp
DDS_ReturnCode_t trigger_flow ()
```

An on-demand FlowController can be created with a DDS_DURATION_INFINITE as **period**, in which case the only trigger source is external (i.e. the FlowController is solely triggered by the user on demand).

**trigger_flow()** can be called on both a strict on-demand FlowController and a hybrid FlowController (internally and externally triggered).

7.6.10 Other FlowController Operations

If you have the FlowController object and need its name, call the FlowController’s **get_name()** operation:

```cpp
const char* DDSFlowController::get_name ( )
```

Conversely, if you have the name of the FlowController and need the FlowController object, call the **DomainParticipant’s lookup_flowcontroller()** operation:

```cpp
DDSFlowController* lookup_flowcontroller (const char * name)
```

To get a FlowController’s **DomainParticipant**, call the FlowController’s **get_participant()** operation:

```cpp
DDSDomainParticipant* get_participant ( )
```

**Note:** It is not safe to lookup a FlowController description while another thread is creating that FlowController.
Chapter 8 Receiving Data

This section discusses how to create, configure, and use Subscribers and DataReaders to receive data. It describes how these objects interact, as well as the types of operations that are available for them.

The goal of this section is to help you become familiar with the Entities you need for receiving data. For up-to-date details such as formal parameters and return codes on any mentioned operations, please see the Connext DDS API Reference HTML documentation.

8.1 Preview: Steps to Receiving Data

There are three ways to receive data:

- Your application can explicitly check for new data by calling a DataReader’s read() or take() operation. This method is also known as polling for data.

- Your application can be notified asynchronously whenever new DDS data samples arrive—this is done with a Listener on either the Subscriber or the DataReader. Connext DDS will invoke the Listener’s callback routine when there is new data. Within the callback routine, user code can access the data by calling read() or take() on the DataReader. This method is the way for your application to receive data with the least amount of latency.

- Your application can wait for new data by using Conditions and a WaitSet, then calling wait(). Connext DDS will block your application’s thread until the criteria (such as the arrival of DDS samples, or a specific status) set in the Condition becomes true. Then your application resumes and can access the data with read() or take().

The DataReader’s read() operation gives your application a copy of the data and leaves the data in the DataReader’s receive queue. The DataReader’s take() operation removes data from the receive queue before giving it to your application.

See 8.4 Using DataReaders to Access Data (Read & Take) on page 553 for details on using DataReaders to access received data.
See 4.6 Conditions and WaitSets on page 202 for details on using Conditions and WaitSets.

To prepare to receive data, create and configure the required Entities:

1. Create a DomainParticipant.
2. Register user data types\(^1\) with the DomainParticipant. For example, the ‘FooDataType’.
3. Use the DomainParticipant to create a Topic with the registered data type.
4. Optionally\(^2\), use the DomainParticipant to create a Subscriber.
5. Use the Subscriber or DomainParticipant to create a DataReader for the Topic.
6. Use a type-safe method to cast the generic DataReader created by the Subscriber to a type-specific DataReader. For example, ‘FooDataReader’.

Then use one of the following mechanisms to receive data.

- To receive DDS data samples by polling for new data:
  - Using a FooDataReader, use the read() or take() operations to access the DDS data samples that have been received and stored for the DataReader. These operations can be invoked at any time, even if the receive queue is empty.

- To receive DDS data samples asynchronously:
  - Install a Listener on the DataReader or Subscriber that will be called back by an internal Connect next DDS thread when new DDS data samples arrive for the DataReader.

1. Create a DDSDataReaderListener for the FooDataReader or a DDSSubscriberListener for Subscriber. In C++, C++/CLI, C# and Java, you must derive your own Listener class from those base classes. In C, you must create the individual functions and store them in a structure.

   If you created a DDSDataReaderListener with the on_data_available() callback enabled: on_data_available() will be called when new data arrives for that DataReader.

   If you created a DDSSubscriberListener with the on_data_on_readers() callback enabled: on_data_on_readers() will be called when data arrives for any DataReader created by the Subscriber.

2. Install the Listener on either the FooDataReader or Subscriber.

   For the DataReader, the Listener should be installed to handle changes in the DATA AVAILABLE status.

\(^1\) Type registration is not required for built-in types (see 3.2.1 Registering Built-in Types on page 39).

\(^2\) You are not required to explicitly create a Subscriber; instead, you can use the ‘implicit Subscriber’ created from the DomainParticipant. See 8.2.1 Creating Subscribers Explicitly vs. Implicitly on page 490.
For the Subscriber, the Listener should be installed to handle changes in the DATA_ON_READERS status.

3. Only 1 Listener will be called back when new data arrives for a DataReader.

Connext DDS will call the Subscriber’s Listener if it is installed. Otherwise, the DataReader’s Listener is called if it is installed. That is, the on_data_on_readers() operation takes precedence over the on_data_available() operation.

If neither Listeners are installed or neither Listeners are enabled to handle their respective statuses, then Connext DDS will not call any user functions when new data arrives for the DataReader.

4. In the on_data_available() method of the DDSDataReaderListener, invoke read() or take() on the FooDataReader to access the data.

If the on_data_on_readers() method of the DDSSubscriberListener is called, the code can invoke read() or take() directly on the Subscriber’s DataReaders that have received new data. Alternatively, the code can invoke the Subscriber’s notify_datareaders() operation. This will in turn call the on_data_available() methods of the DataReaderListeners (if installed and enabled) for each of the DataReaders that have received new DDS data samples.

To wait (block) until DDS data samples arrive:

1. Use the DataReader to create a ReadCondition that describes the DDS samples for which you want to wait. For example, you can specify that you want to wait for never-before-seen DDS samples from DataReaders that are still considered to be ‘alive.’

Alternatively, you can create a StatusCondition that specifies you want to wait for the ON_DATA_AVAILABLE status.

2. Create a WaitSet.

3. Attach the ReadCondition or StatusCondition to the WaitSet.

4. Call the WaitSet’s wait() operation, specifying how long you are willing to wait for the desired DDS samples. When wait() returns, it will indicate that it timed out, or that the attached Condition become true (and therefore the desired DDS samples are available).

5. Using a FooDataReader, use the read() or take() operations to access the DDS data samples that have been received and stored for the DataReader.

8.2 Subscribers

An application that intends to subscribe to information needs the following Entities: DomainParticipant, Topic, Subscriber, and DataReader. All Entities have a corresponding specialized Listener and a set of QosPolicies. The Listener is how Connext DDS notifies your application of status changes relevant to the Entity. The QosPolicies allow your application to configure the behavior and resources of the Entity.
8.2 Subscribers

- The *DomainParticipant* defines the DDS domain on which the information will be available.
- The *Topic* defines the name of the data to be subscribed, as well as the type (format) of the data itself.
- The *DataReader* is the Entity used by the application to subscribe to updated values of the data. The *DataReader* is bound at creation time to a *Topic*, thus specifying the named and typed data stream to which it is subscribed. The application uses the *DataReader’s* `read()` or `take()` operation to access DDS data samples received for the *Topic*.
- The *Subscriber* manages the activities of several *DataReader* entities. The application receives data using a *DataReader* that belongs to a *Subscriber*. However, the *Subscriber* will determine when the data received from applications is actually available for access through the *DataReader*. Depending on the settings of various QosPolicies of the *Subscriber* and *DataReader*, data may be buffered until DDS data samples for associated *DataReaders* are also received. By default, the data is available to the application as soon as it is received.

For more information, see 8.2.1 Creating Subscribers Explicitly vs. Implicitly on page 490.

The UML diagram in Figure 8.1: Subscription Module on the next page shows how these *Entities* are related as well as the methods defined for each Entity.

Subscribers are used to perform the operations listed in Table 8.1 Subscriber Operations. For details such as formal parameters and return codes, please see the API Reference HTML documentation. Otherwise, you can find more information about the operations by looking in the section listed under the Reference on page 489 column.
Note: Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.
# 8.2 Subscribers

## Table 8.1 Subscriber Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataReaders</td>
<td>begin_access</td>
<td>Indicates that the application is about to access the DDS data samples in the <strong>DataReaders</strong> of the <strong>Subscriber</strong>.</td>
<td>8.2.5 Beginning and Ending Group-Ordered Access on page 499</td>
</tr>
<tr>
<td></td>
<td>create_datareader</td>
<td>Creates a <strong>DataReader</strong>.</td>
<td>8.3.1 Creating DataReaders on page 510</td>
</tr>
<tr>
<td></td>
<td>create_datareader_with_profile</td>
<td>Creates a <strong>DataReader</strong> with QoS from a specified QoS profile.</td>
<td>8.2.4.6 Subscriber QoS-Related Operations on page 498</td>
</tr>
<tr>
<td></td>
<td>copy_from_topic_qos</td>
<td>Copies relevant QosPolicies from a <strong>Topic</strong> into a <strong>DataReaderQos</strong> structure.</td>
<td></td>
</tr>
<tr>
<td>DataReaders cont'd</td>
<td>delete_contained_entities</td>
<td>Deletes all the <strong>DataReaders</strong> that were created by the <strong>Subscriber</strong>. Also deletes the corresponding <strong>ReadConditions</strong> created by the contained <strong>DataReaders</strong>.</td>
<td>8.2.3.1 Deleting Contained DataReaders on page 493</td>
</tr>
<tr>
<td></td>
<td>delete_datareader</td>
<td>Deletes a specific <strong>DataReader</strong>.</td>
<td>8.3.3 Deleting DataReaders on page 512</td>
</tr>
<tr>
<td></td>
<td>end_access</td>
<td>Indicates that the application is done accessing the DDS data samples in the <strong>DataReaders</strong> of the <strong>Subscriber</strong>.</td>
<td>8.2.5 Beginning and Ending Group-Ordered Access on page 499</td>
</tr>
<tr>
<td></td>
<td>get_all_datareaders</td>
<td>Retrieves all the <strong>DataReaders</strong> created from this <strong>Subscriber</strong>.</td>
<td>8.3.2 Getting All DataReaders on page 512</td>
</tr>
<tr>
<td></td>
<td>get_datareaders</td>
<td>Returns a list of <strong>DataReaders</strong> that contain DDS samples with the specified <code>sample_states</code>, <code>view_states</code> and <code>instance_states</code>.</td>
<td>8.2.7 Getting DataReaders with Specific DDS Samples on page 502</td>
</tr>
<tr>
<td></td>
<td>get_default_datareader_qos</td>
<td>Copies the <strong>Subscriber</strong>'s default <strong>DataReaderQos</strong> values into a <strong>DataReaderQos</strong> structure.</td>
<td>8.2.4 Setting Subscriber QosPolicies on page 493</td>
</tr>
<tr>
<td>DataReaders cont'd</td>
<td>get_status_changes</td>
<td>Gets all status changes.</td>
<td>4.1.4 Getting Status and Status Changes on page 171</td>
</tr>
<tr>
<td></td>
<td>lookup_datareader</td>
<td>Retrieves a <strong>DataReader</strong> previously created for a specific <strong>Topic</strong>.</td>
<td>8.2.8 Finding a Subscriber's Related Entities on page 502</td>
</tr>
<tr>
<td></td>
<td>notify_datareaders</td>
<td>Invokes the <code>on_data_available()</code> operation for attached <strong>Listeners</strong> of <strong>DataReaders</strong> that have new DDS data samples.</td>
<td>8.2.6 Setting Up Subscriber Listeners on page 500</td>
</tr>
<tr>
<td></td>
<td>set_default_datareader_qos</td>
<td>Sets or changes the <strong>Subscriber</strong>'s default <strong>DataReaderQoS</strong> values.</td>
<td>8.2.4 Setting Subscriber QosPolicies on page 493</td>
</tr>
</tbody>
</table>
8.2.1 Creating Subscribers Explicitly vs. Implicitly

To receive data, your application must have a Subscriber. However, you are not required to explicitly create a Subscriber. If you do not create one, the middleware will implicitly create a Subscriber the first time you create a DataReader using the DomainParticipant’s operations. It will be created with default QoS (DDS_SUBSCRIBER_QOS_DEFAULT) and no Listener. The 'implicit Subscriber' can be

---

**Table 8.1 Subscriber Operations**

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries and Profiles</td>
<td>get_default_library</td>
<td>Gets the Subscriber’s default QoS profile library.</td>
<td>8.2.4.4 Getting and Settings Subscriber’s Default QoS Profile and Library on page 497</td>
</tr>
<tr>
<td></td>
<td>get_default_profile</td>
<td>Gets the Subscriber’s default QoS profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_profile_library</td>
<td>Gets the library that contains the Subscriber’s default QoS profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_library</td>
<td>Sets the default library for a Subscriber.</td>
<td>8.2.8 Finding a Subscriber’s Related Entities on page 502</td>
</tr>
<tr>
<td></td>
<td>set_default_profile</td>
<td>Sets the default profile for a Subscriber.</td>
<td></td>
</tr>
<tr>
<td>Participants</td>
<td>get_participant</td>
<td>Gets the Subscriber’s DomainParticipant.</td>
<td>4.1.2 Enabling DDS Entities on page 168</td>
</tr>
<tr>
<td></td>
<td>enable</td>
<td>Enables the Subscriber.</td>
<td>8.2.4.2 Comparing QoS Values on page 496</td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Compares two Subscriber’s QoS structures for equality.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_listener</td>
<td>Gets the currently installed Listener.</td>
<td>8.2.6 Setting Up SubscriberListeners on page 500</td>
</tr>
<tr>
<td>Subscribers</td>
<td>get_qos</td>
<td>Gets the Subscriber’s current QosPolicy settings. This is most often used in preparation for calling set_qos.</td>
<td>8.2.4.3 Changing QoS Settings After Subscriber Has Been Created on page 496</td>
</tr>
<tr>
<td></td>
<td>set_listener</td>
<td>Sets the Subscriber’s Listener. If you created the Subscriber without a Listener, you can use this operation to add one later.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_qos</td>
<td>Sets the Subscriber’s QoS. You can use this operation to change the values for the Subscriber’s QosPolicies. Note, however, that not all QosPolicies can be changed after the Subscriber has been created.</td>
<td>8.2.4.3 Changing QoS Settings After Subscriber Has Been Created on page 496</td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>Sets the Subscriber’s QoS based on a QoS profile.</td>
<td>8.2.4.3 Changing QoS Settings After Subscriber Has Been Created on page 496</td>
</tr>
</tbody>
</table>
accessed using the DomainParticipant’s `get_implicit_subscriber()` operation (see 9.3.10 Getting the Implicit Publisher or Subscriber on page 629). You can use this ‘implicit Subscriber’ just like any other Subscriber (it has the same operations, QosPolicies, etc.). So you can change the mutable QoS and set a Listener if desired.

A Subscriber (implicit or explicit) gets its own default QoS and the default QoS for its child DataReaders from the DomainParticipant. These default QoS are set when the Subscriber is created. (This is true for Publishers and DataWriters, too.)

DataReaders are created by calling `create_datareader()` or `create_datareader_with_profile()`—these operations exist for DomainParticipants and Subscribers. If you use the DomainParticipant to create a DataReader, it will belong to the implicit Subscriber. If you use a Subscriber to create a DataReader, it will belong to that Subscriber.

The middleware will use the same implicit Subscriber for all DataReaders that are created using the DomainParticipant’s operations.

Having the middleware implicitly create a Subscriber allows you to skip the step of creating a Subscriber. However, having all your DataReaders belong to the same Subscriber can reduce the concurrency of the system because all the read operations will be serialized.

### 8.2.2 Creating Subscribers

Before you can explicitly create a Subscriber, you need a DomainParticipant (9.3 DomainParticipants on page 609). To create a Subscriber, use the DomainParticipant’s `create_subscriber()` or `create_subscriber_with_profile()` operation.

A QoS profile is a way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see Configuring QoS with XML (Chapter 19 on page 848).

**Note:** The Modern C++ API provides Subscriber constructors whose first, and only required argument is the DomainParticipant.

```cpp
DDSSubscriber* create_subscriber ( const DDS_SubscriberQos &qos, 
                                DDS_SubscriberListener * listener, 
                                DDS_StatusMask mask )

DDSSubscriber* create_subscriber_with_profile ( const char * library_name, 
                                               const char * profile_name, 
                                               DDS_SubscriberListener * listener, 
                                               DDS_StatusMask mask )
```

Where:

1In the Modern C++ API, you always use a DataReader constructor.
8.2.3 Deleting Subscribers

If you want the default QoS settings (described in the API Reference HTML documentation), use DDS_SUBSCRIBER_QOS_DEFAULT for this parameter (see Figure 8.2: Creating a Subscriber with Default QosPolicies below). If you want to customize any of the QoS policies, supply a QoS structure (see Figure 8.3: Creating a Subscriber with Non-Default QosPolicies (not from a profile) on page 495). The QoS structure for a Subscriber is described in 8.5 Subscriber QosPolicies on page 570.

Note: If you use DDS_SUBSCRIBER_QOS_DEFAULT, it is not safe to create the Subscriber while another thread may be simultaneously calling set_default_subscriber_qos().

Listeners are callback routines. Connext DDS uses them to notify your application when specific events (new DDS data samples arrive and status changes) occur with respect to the Subscriber or the DataReaders created by the Subscriber. The listener parameter may be set to NULL if you do not want to install a Listener. If you use NULL, the Listener of the DomainParticipant to which the Subscriber belongs will be used instead (if it is set). For more information on SubscriberListeners, see 8.2.6 Setting Up SubscriberListeners on page 500.

This bit-mask indicates which status changes will cause the Subscriber's Listener to be invoked. The bits set in the mask must have corresponding callbacks implemented in the Listener. If you use NULL for the Listener, use DDS_STATUS_MASK_NONE for this parameter. If the Listener implements all callbacks, use DDS_STATUS_MASK_ALL. For information on Status, see 4.4 Listeners on page 189.

This bit-mask indicates which status changes will cause the Subscriber's Listener to be invoked. The bits set in the mask must have corresponding callbacks implemented in the Listener. If you use NULL for the Listener, use DDS_STATUS_MASK_NONE for this parameter. If the Listener implements all callbacks, use DDS_STATUS_MASK_ALL. For information on Status, see 4.4 Listeners on page 189.

A QoS Library is a named set of QoS profiles. See 19.3 QoS Profiles on page 850.

A QoS profile groups a set of related QoS, usually one per entity. See 19.3 QoS Profiles on page 850.

Figure 8.2: Creating a Subscriber with Default QosPolicies

```cpp
// create the subscriber
DDSSubscriber* subscriber =
    participant->create_subscriber(
        DDS_SUBSCRIBER_QOS_DEFAULT,
        NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
    // handle error
}
```

For more examples, see 8.2.4.1 Configuring QoS Settings when the Subscriber is Created on page 494.

After you create a Subscriber, the next step is to use the Subscriber to create a DataReader for each Topic, see 8.3.1 Creating DataReaders on page 510. For a list of operations you can perform with a Subscriber, see Table 8.1 Subscriber Operations.

8.2.3 Deleting Subscribers

(Note: in the Modern C++ API, Entities are automatically destroyed, see 4.1.1 Creating and Deleting DDS Entities on page 167)

This section applies to both implicitly and explicitly created Subscribers.

To delete a Subscriber:
1. You must first delete all *DataReaders* that were created with the *Subscriber*. Use the *Subscriber’s delete_datareader()* operation (8.3.1 Creating DataReaders on page 510) to delete them one at a time, or use the *delete_contained_entities()* operation (8.2.3.1 Deleting Contained DataReaders below) to delete them all at the same time.

   ```c
   DDS_ReturnCode_t delete_datareader (DDSDataReader *a_datareader)
   ```

2. Delete the *Subscriber* by using the DomainParticipant’s *delete_subscriber()* operation ()

   **Note:** A *Subscriber* cannot be deleted within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

### 8.2.3.1 Deleting Contained DataReaders

The *Subscriber’s delete_contained_entities()* operation deletes all the *DataReaders* that were created by the *Subscriber*. It also deletes the ReadConditions created by each contained *DataReader*.

   ```c
   DDS_ReturnCode_t DDSSubscriber::delete_contained_entities ()
   ```

After this operation returns successfully, the application may delete the *Subscriber* (see 8.2.3 Deleting Subscribers on the previous page).

The operation will return PRECONDITION_NOT_MET if any of the contained entities cannot be deleted. This will occur, for example, if a contained *DataReader* cannot be deleted because the application has called *read()* but has not called the corresponding *return_loan()* operation to return the loaned DDS samples.

### 8.2.4 Setting Subscriber QosPolicies

A *Subscriber’s* QosPolicies control its behavior. Think of the policies as the configuration and behavior ‘properties’ for the *Subscriber*. The *DDS_SubscriberQos* structure has the following format:

```c
struct DDS_SubscriberQos {
    DDS_PresentationQosPolicy             presentation;
    DDS_PartitionQosPolicy                partition;
    DDS_GroupDataQosPolicy                group_data;
    DDS_EntityFactoryQosPolicy            entity_factory;
    DDS_ExclusiveAreaQosPolicy            exclusive_area;
    DDS_EntityNameQosPolicy               subscriber_name;
};
```

**Note:** *set_qos()* cannot always be used by a *Listener*, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

Table 8.2 Subscriber QosPolicies summarizes the meaning of each policy. *Subscribers* have the same set of QosPolicies as Publishers; they are described in detail in 7.4 Publisher/Subscriber QosPolicies on page 345. For information on why you would want to change a particular QosPolicy, see the referenced
section. For defaults and valid ranges, please refer to the API Reference HTML documentation for each policy.

### Table 8.2 Subscriber QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.4.2 ENTITYFACTORY QosPolicy on page 348</td>
<td>Whether or not new entities created from this entity will start out as ‘enabled.’</td>
</tr>
<tr>
<td>7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418</td>
<td>Assigns a name and role_name to a Subscriber.</td>
</tr>
<tr>
<td>7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) on page 350</td>
<td>Whether or not the entity uses a multi-thread safe region with deadlock protection.</td>
</tr>
<tr>
<td>7.4.4 GROUP_DATA QosPolicy on page 353</td>
<td>A place to pass group-level information among applications. Usage is application-dependent.</td>
</tr>
<tr>
<td>7.4.5 PARTITION QosPolicy on page 356</td>
<td>Set of strings that introduces a logical partition among Topics visible by Publisher/Subscriber.</td>
</tr>
<tr>
<td>7.4.6 PRESENTATION QosPolicy on page 362</td>
<td>The order in which instance changes are presented to the Subscriber. By default, no order is used.</td>
</tr>
</tbody>
</table>

#### 8.2.4.1 Configuring QoS Settings when the Subscriber is Created

As described in 8.2.2 Creating Subscribers on page 491, there are different ways to create a Subscriber, depending on how you want to specify its QoS (with or without a QoS Profile).

- In 8.2.2 Creating Subscribers on page 491 is an example of how to explicitly create a Subscriber with default QosPolicies. It used the special constant, DDS_SUBSCRIBER_QOS_DEFAULT, which indicates that the default QoS values for a Subscriber should be used. The default Subscriber QosPolicies are configured in the DomainParticipant; you can change them with the DomainParticipant’s set_default_subscriber_qos() or set_default_subscriber_qos_with_profile() operation (see 9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627).

- To create a Subscriber with non-default QoS settings, without using a QoS profile, see Figure 8.3: Creating a Subscriber with Non-Default QosPolicies (not from a profile) on the next page. It uses the DomainParticipant’s get_default_subscriber_qos() method to initialize a DDS_SubscriberQos structure. Then the policies are modified from their default values before the QoS structure is passed to create_subscriber().

- You can also create a Subscriber and specify its QoS settings via a QoS Profile. To do so, call create_subscriber_with_profile(), as seen in Figure 8.4: Creating a Subscriber with a QoS Profile on the next page.

- If you want to use a QoS profile, but then make some changes to the QoS before creating the Subscriber, call get_subscriber_qos_from_profile(), modify the QoS and use the modified QoS struc-
8.2.4 Setting Subscriber QosPolicies

Setting Subscriber QosPolicies when calling `create_subscriber()`, as seen in Figure 8.5: Getting QoS Values from a Profile, Changing QoS Values, Creating a Subscriber with Modified QoS Values on the next page.

For more information, see 8.2.2 Creating Subscribers on page 491 and Configuring QoS with XML (Chapter 19 on page 848).

**Figure 8.3: Creating a Subscriber with Non-Default QosPolicies (not from a profile)**

```c
DDS_SubscriberQos subscriber_qos;
// get defaults
if (participant->get_default_subscriber_qos(subscriber_qos) != DDS_RETCODE_OK) {
    // handle error
}
// make QoS changes here. for example, this changes the ENTITY_FACTORY QoS
subscriber_qos.entity_factory.autoenable_created_entities=DDS_BOOLEAN_FALSE;
// create the subscriber
DDSSubscriber * subscriber = participant->create_subscriber(subscriber_qos, NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
    // handle error
}
```

**Figure 8.4: Creating a Subscriber with a QoS Profile**

```c
// create the subscriber with QoS profile
DDSSubscriber * subscriber = participant->create_subscriber_with_profile(
    "MySubscriberLibrary", "MySubscriberProfile", NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
    // handle error
}
```

---

Note: In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.
8.2.4 Setting Subscriber QosPolicies

Figure 8.5: Getting QoS Values from a Profile, Changing QoS Values, Creating a Subscriber with Modified QoS Values

```c
DDS_SubscriberQos subscriber_qos;
// Get subscriber QoS from profile
retcode = factory->get_subscriber_qos_from_profile(subscriber_qos,
    "SubscriberLibrary", "SubscriberProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
// for example, this changes the ENTITY_FACTORY QoS
subscriber_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_TRUE;
// create the subscriber with modified QoS
DDSPublisher* subscriber = participant->create_subscriber(
    "Example Foo", type_name, subscriber_qos,
    NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
    // handle error
}
```

8.2.4.2 Comparing QoS Values

The `equals()` operation compares two `Subscriber`'s DDS_SubscriberQos structures for equality. It takes two parameters for the two `Subscriber`'s QoS structures to be compared, then returns TRUE if they are equal (all values are the same) or FALSE if they are not equal.

8.2.4.3 Changing QoS Settings After Subscriber Has Been Created

There are 2 ways to change an existing `Subscriber`'s QoS after it is has been created—again depending on whether or not you are using a QoS Profile.

- To change an existing `Subscriber`'s QoS programmatically (that is, without using a QoS profile), `get_qos()` and `set_qos()`. See the example code in Figure 8.6: Changing the Qos of an Existing Subscriber on the next page. It retrieves the current values by calling the `Subscriber`'s `get_qos()` operation. Then it modify the value and call `set_qos()` to apply the new value. Note, however, that some QosPolicies cannot be changed after the `Subscriber` has been enabled—this restriction is noted in the descriptions of the individual QosPolicies.

- You can also change a `Subscriber`'s (and all other Entities') QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see Figure 8.7: Changing the QoS of an Existing Subscriber with a QoS Profile on the next page. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

---

1 Note: In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.
8.2.4 Setting Subscriber QosPolicies

**Figure 8.6: Changing the Qos of an Existing Subscriber**

```c
DDS_SubscriberQos subscriber_qos;
// Get current QoS. subscriber points to an existing DDSSubscriber.
if (subscriber->get_qos(subscriber_qos) != DDS_RETCODE_OK) {
    // handle error
}
// make changes
// New entity_factory autoenable_created_entities will be true
subscriber_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_TRUE;
// Set the new QoS
if (subscriber->set_qos(subscriber_qos) != DDS_RETCODE_OK) {
    // handle error
}
```

**Figure 8.7: Changing the Qos of an Existing Subscriber with a QoS Profile**

```c
retcode = subscriber->set_qos_with_profile(
    "SubscriberProfileLibrary";"SubscriberProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

### 8.2.4.4 Getting and Settings Subscriber’s Default QoS Profile and Library

You can retrieve the default QoS profile used to create Subscribers with the `get_default_profile()` operation. You can also get the default library for Subscribers, as well as the library that contains the Subscriber’s default profile (these are not necessarily the same library); these operations are called `get_default_library()` and `get_default_library_profile()`, respectively. These operations are for informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see Configuring QoS with XML (Chapter 19 on page 848).

```c
virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_profile_library ()
```

There are also operations for setting the Subscriber’s default library and profile:

```c
DDS_ReturnCode_t set_default_library (const char * library_name)
DDS_ReturnCode_t set_default_profile (const char * library_name,
                                    const char * profile_name)
```

These operations only affect which library/profile will be used as the default the next time a default Subscriber library/profile is needed during a call to one of this Subscriber’s operations.

When calling a Subscriber operation that requires a `profile_name` parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

If the default library/profile is not set, the Subscriber inherits the default from the DomainParticipant.
set_default_profile() does not set the default QoS for DataReaders created by the Subscriber; for this functionality, use the Subscriber’s set_default_datatreader_qos_with_profile(), see 8.2.4.5 Getting and Setting Default QoS for DataReaders below (you may pass in NULL after having called the Subscriber’s set_default_profile()).

set_default_profile() does not set the default QoS for newly created Subscribers; for this functionality, use the DomainParticipant’s set_default_subscriber_qos_with_profile() operation, see 9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627.

8.2.4.5 Getting and Setting Default QoS for DataReaders

These operations set the default QoS that will be used for new DataReaders if create_datatreader() is called with DDS_DATAREADER_QOS_DEFAULT as the ‘qos’ parameter:

```c
DD_ReturnCode_t set_default_datatreader_qos (const DDS_DATAReaderQos &qos)
```

The above operations may potentially allocate memory, depending on the sequences contained in some QoS policies.

To get the default QoS that will be used for creating DataReaders if create_datatreader() is called with DDS_DATAREADER_QOS_DEFAULT as the ‘qos’ parameter:

```c
DD_ReturnCode_t get_default_datatreader_qos (DDS_DATAReaderQos & qos)
```

The above operation gets the QoS settings that were specified on the last successful call to set_default_datatreader_qos() or set_default_datatreader_qos_with_profile(), or if the call was never made, the default values listed in DDS_DATAReaderQos.

**Note:** It is not safe to set the defaultDataReader QoS values while another thread may be simultaneously calling get_default_datatreader_qos(), set_default_datatreader_qos() or create_datatreader() with DDS_DATAREADER_QOS_DEFAULT as the qos parameter. It is also not safe to get the default DataReader QoS values while another thread may be simultaneously calling set_default_datatreader_qos().

8.2.4.6 Subscriber QoS-Related Operations

- **Copying a Topic’s QoS into a DataReader’s QoS**

  This method is provided as a convenience for setting the values in a DataReaderQos structure before using that structure to create a DataReader. As explained in 5.1.3 Setting Topic QosPolicies on page 219, most of the policies in a TopicQos structure do not apply directly to the Topic itself, but to the associated DataWriters and DataReaders of that Topic. The TopicQos serves as a single container where the values of QosPolicies that must be set compatibly across matching DataWriters and DataReaders can be stored.
Thus instead of setting the values of the individual QosPolicies that make up a `DataReaderQos` structure every time you need to create a `DataReader` for a `Topic`, you can use the Subscriber’s `copy_from_topic_qos()` operation to “import” the `Topic`’s QosPolicies into a `DataReaderQos` structure. This operation copies the relevant policies in the `TopicQos` to the corresponding policies in the `DataReaderQos`.

This copy operation will often be used in combination with the Subscriber’s `get_default_datareader_qos()` and the `Topic`’s `get_qos()` operations. The `Topic`’s QoS values are merged on top of the Subscriber’s default `DataReader` QosPolicies with the result used to create a new `DataReader`, or to set the QoS of an existing one (see 8.3.9 Setting DataReader QosPolicies on page 543).

- **Copying a Subscriber’s QoS**

  In the C API users should use the `DDS_SubscriberQos_copy()` operation rather than using structure assignment when copying between two QoS structures. The `copy()` operation will perform a deep copy so that policies that allocate heap memory such as sequences are copied correctly. In C++, C++/CLI, C# and Java, a copy constructor is provided to take care of sequences automatically.

- **Clearing QoS-Related Memory**

  Some QosPolicies contain sequences that allocate memory dynamically as they grow or shrink. The C API’s `DDS_SubscriberQos_finalize()` operation frees the memory used by sequences but otherwise leaves the QoS unchanged. C users should call `finalize()` on all `DDS_SubscriberQos` objects before they are freed, or for QoS structures allocated on the stack, before they go out of scope. In C++, C++/CLI, C# and Java, the memory used by sequences is freed in the destructor.

### 8.2.5 Beginning and Ending Group-Ordered Access

The Subscriber’s `begin_access()` operation indicates that the application is about to access the DDS data samples in any of the `DataReaders` attached to the Subscriber.

If the Subscriber’s `access_scope` (in the 7.4.6 PRESENTATION QosPolicy on page 362) is GROUP or HIGHEST_OFFERED and `ordered_access` (also in the 7.4.6 PRESENTATION QosPolicy on page 362) is TRUE, the application is required to use this operation to access the DDS samples in order across `DataWriters` of the same group (Publisher with `access_scope` GROUP).

In the above case, `begin_access()` must be called prior to calling any of the sample-accessing operations: `get_datareaders()` on the Subscriber, and `read()`, `take()`, `read_w_condition()`, and `take_w_condition()` on any `DataReader`.

Once the application has finished accessing the DDS data samples, it must call `end_access()`. To see how to read samples in order when the Subscriber’s `begin_access()` operation is called, see 8.2.7 Getting DataReaders with Specific DDS Samples on page 502.
8.2.6 Setting Up SubscriberListeners

The application is not required to call `begin_access()` and `end_access()` to access the DDS samples in order if the Publisher’s `access_scope` is something other than GROUP. In this case, calling `begin_access()` and `end_access()` is not considered an error and has no effect.

Calls to `begin_access()` and `end_access()` may be nested and must be balanced. That is, `end_access()` close a previous call to `begin_access()`.

### 8.2.6 Setting Up SubscriberListeners

Like all Entities, Subscribers may optionally have Listeners. Listeners are user-defined objects that implement a DDS-defined interface (i.e. a pre-defined set of callback functions). Listeners provide the means for Connext DDS to notify applications of any changes in Statuses (events) that may be relevant to it. By writing the callback functions in the Listener and installing the Listener into the Subscriber, applications can be notified to handle the events of interest. For more general information on Listeners and Statuses, see 4.4 Listeners on page 189.

Note: Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

As illustrated in Figure 8.1: Subscription Module on page 488, the SubscriberListener interface extends the DataReaderListener interface. In other words, the SubscriberListener interface contains all the functions in the DataReaderListener interface. In addition, a SubscriberListener has an additional function: `on_data_on_readers()`, corresponding to the Subscriber’s `DATA_ON_READERS` status. This is the only status that is specific to a Subscriber. This status is closely tied to the `DATAAVAILABLE` status (8.3.7.1 `DATAAVAILABLE` Status on page 517) of DataReaders.

The Subscriber’s `DATA_ON_READERS` status is set whenever the `DATAAVAILABLE` status is set for any of the DataReaders created by the Subscriber. This implies that one of its DataReaders has received new DDS data samples. When the `DATA_ON_READERS` status is set, the SubscriberListener’s `on_data_on_readers()` method will be invoked.

The `DATA_ON_READERS` status of a Subscriber takes precedence over the `DATAAVAILABLE` status of any of its DataReaders. Thus, when data arrives for a DataReader, the `on_data_on_readers()` operation of the SubscriberListener will be called instead of the `on_data_available()` operation of the DataReaderListener—assuming that the Subscriber has a Listener installed that is enabled to handle changes in the `DATA_ON_READERS` status. (Note however, that in the SubscriberListener’s `on_data_on_readers()` operation, you may choose to call `notify_datareaders()`, which in turn may cause the DataReaderListener’s `on_data_available()` operation to be called.)

All of the other methods of a SubscriberListener will be called back for changes in the Statuses of Subscriber’s DataReaders only if the DataReader is not set up to handle the statuses itself.

If you want a Subscriber to handle status events for its DataReaders, you can set up a SubscriberListener during the Subscriber’s creation or use the `set_listener()` method after the Subscriber is created. The last parameter is a bit-mask with which you should set which Status events that the SubscriberListener will handle. For example,
As previously mentioned, the callbacks in the SubscriberListener act as ‘default’ callbacks for all the DataReaders contained within. When Connext DDS wants to notify a DataReader of a relevant Status change (for example, SUBSCRIPTION_MATCHED), it first checks to see if the DataReader has the corresponding DataReaderListener callback enabled (such as the on_subscription_matched() operation). If so, Connext DDS dispatches the event to the DataReaderListener callback. Otherwise, Connext DDS dispatches the event to the corresponding SubscriberListener callback.

NOTE, the reverse is true for the DATA_ON_READERS/DATA_AVAILABLE status. When DATA_AVAILABLE changes for any DataReaders of a Subscriber, Connext DDS first checks to see if the SubscriberListener has DATA_ON_READERS enabled. If so, Connext DDS will invoke the on_data_on_readers() callback. Otherwise, Connext DDS dispatches the event to the Listener (on_data_available()) of the DataReader whose DATA_AVAILABLE status actually changed.

A particular callback in a DataReader is not enabled if either:

- The application installed a NULL DataReaderListener (meaning there are no callbacks for the DataReader at all).
- The application has disabled the callback for a DataReaderListener. This is done by turning off the associated status bit in the mask parameter passed to the set_listener() or create_datareader() call when installing the DataReaderListener on the DataReader. For more information on DataReaderListener, see 8.3.4 Setting Up DataReaderListeners on page 512.

Similarly, the callbacks in the DomainParticipantListener act as ‘default’ callbacks for all the Subscribers that belong to it. For more information on DomainParticipantListeners, see 9.3.6 Setting Up DomainParticipantListeners on page 620.

The Subscriber also provides an operation called notify_datareaders() that can be used to invoke the on_data_available() callbacks of DataReaders who have new DDS data samples in their receive queues. Often notify_datareaders() will be used in the on_data_on_readers() callback to pass off the real processing of data from the SubscriberListener to the individual DataReaderListeners.

Calling notify_datareaders() causes the DATA_ON_READERS status to be reset.

Figure 8.8: Simple SubscriberListener on the next page shows a SubscriberListener that simply notifies its DataReaders when new data arrives.
8.2.7 Getting DataReaders with Specific DDS Samples

The Subscriber’s get_datareaders() operation retrieves a list of DataReaders that have DDS samples with specific sample_states, view_states, and instance_states.

If the application is outside a begin_access/end_access() block, or if the Subscriber’s access_scope (in the 7.4.6 PRESENTATION QosPolicy on page 362) is INSTANCE or TOPIC, or ordered_access (also in the 7.4.6 PRESENTATION QosPolicy on page 362) is FALSE, the returned collection is a 'set' containing each DataReader at most once, in no specified order.

If the application is within a begin_access/end_access() block, and the Subscriber’s access_scope is GROUP or HIGHEST_OFFERED, and ordered_access is TRUE, the returned collection is a 'list' of DataReaders, where a DataReader may appear more than one time.

To retrieve the DDS samples in the order in which they were published across DataWriters of the same group (a Publisher configured with GROUP access_scope), the application should read/take() from each DataReader in the same order as appears in the output sequence. The application will move to the next DataReader when the read/take() operation fails with NO_DATA.

For more information, see 8.4.6 The SampleInfo Structure on page 565.

8.2.8 Finding a Subscriber’s Related Entities

These Subscriber operations are useful for obtaining a handle to related entities:

- get_participant(): Gets the DomainParticipant with which a Subscriber was created.
• **lookup_datareader()**: Finds a *DataReader* created by the *Subscriber* with a *Topic* of a particular name. Note that if multiple *DataReaders* were created by the same *Subscriber* with the same *Topic*, any one of them may be returned by this method.

You can use this operation on a built-in *Subscriber* to access the built-in *DataReaders* for the built-in topics. The built-in *DataReader* is created when this operation is called on a built-in topic for the first time.

If you are going to modify the transport properties for the built-in *DataReaders*, do so before using this operation. Built-in transports are implicitly registered when the *DomainParticipant* is enabled or the first *DataWriter/DataReader* is created. To ensure that built-in *DataReaders* receive all the discovery traffic, you should lookup the *DataReader* before the *DomainParticipant* is enabled. Therefore the suggested sequence when looking up built-in *DataReaders* is:

1. Create a disabled *DomainParticipant* (see 7.4.2 ENTITYFACTORY QosPolicy on page 348).
2. If you want to use non-default values, modify the built-in transport properties (see 16.5 Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties() on page 799).
3. Call **get_builtin_subscriber()** (see 18.2 Built-in DataReaders on page 832).
4. Call **lookup_datareader()**.
5. Call **enable()** on the *DomainParticipant* (see 4.1.2 Enabling DDS Entities on page 168).

• **DDS_Subscriber_as_Entity()**: This method is provided for C applications and is necessary when invoking the parent class *Entity* methods on *Subscribers*. For example, to call the *Entity* method **get_status_changes()** on a *Subscriber*, *my_sub*, do the following:

```
DDS_Entity_get_status_changes(DDS_Subscriber_as_Entity(my_sub))
```

• **DDS_Subscriber_as_Entity()** is not provided in the C++, C++/CLI, C# and Java APIs because the object-oriented features of those languages make it unnecessary.

### 8.2.9 Statuses for Subscribers

The status indicators for a *Subscriber* are the same as those available for its *DataReaders*, with one additional status: **DATA_ON_READERS** (8.2.9.1 DATA_ON_READERS Status on the next page). The following statuses can be monitored by the *SubscriberListener*.

- **8.2.9.1** DATA_ON_READERS Status on the next page
- **8.3.7.1** DATAAVAILABLE Status on page 517
- **8.3.7.4** LIVELINESS_CHANGED Status on page 524
8.3.7.5 REQUESTED_DEADLINE_MISSED Status on page 526
8.3.7.6 REQUESTED_INCOMPATIBLE_QOS Status on page 526
8.3.7.7 SAMPLE_LOST Status on page 527
8.3.7.8 SAMPLE_REJECTED Status on page 531
8.3.7.9 SUBSCRIPTION_MATCHED Status on page 533

You can access Subscriber status by using a SubscriberListener or its inherited get_status_changes() operation (see 4.1.4 Getting Status and Status Changes on page 171), which can be used to explicitly poll for the DATA_ON_READERS status of the Subscriber.

### 8.2.9.1 DATA_ON_READERS Status

The DATA_ON_READERS status, like the DATAAVAILABLE status for DataReaders, is a read communication status, which makes it somewhat different from other plain communication statuses. (See 4.3.1 Types of Communication Status on page 183 for more information on statuses and the difference between read and plain statuses.) In particular, there is no status-specific data structure; the status is either changed or not, there is no additional associated information.

The DATA_ON_READERS status indicates that there is new data available for one or more DataReaders that belong to this Subscriber. The DATAAVAILABLE status for each such DataReader will also be updated.

The DATA_ON_READERS status is reset (the corresponding bit in the bitmask is turned off) when you call read(), take(), or one of their variations on any of the DataReaders that belong to the Subscriber. This is true even if the DataReader on which you call read/take is not the same DataReader that caused the DATA_ON_READERS status to be set in the first place. This status is also reset when you call notify_datareaders() on the Subscriber, or after on_data_on_readers() is invoked.

If a SubscriberListener has both on_data_on_readers() and on_data_available() callbacks enabled (by turning on both status bits), only on_data_on_readers() is called.

### 8.3 DataReaders

To create a DataReader, you need a DomainParticipant, a Topic, and optionally, a Subscriber. You need at least one DataReader for each Topic whose DDS data samples you want to receive.

After you create a DataReader, you will be able to use the operations listed in Table 8.3 DataReader Operations. You are likely to use many of these operations from within your DataReader’s Listener, which is invoked when there are status changes or new DDS data samples. For more details on all operations, see the API reference HTML documentation. The DataReaderListener is described in 8.3.4 Setting Up DataReaderListeners on page 512.

DataReaders are created by using operations on a DomainParticipant or a Subscriber, as described in 8.2.1 Creating Subscribers Explicitly vs. Implicitly on page 490. If you use the DomainParticipant’s
operations, the \textit{DataReader} will belong to an implicit \textit{Subscriber} that is automatically created by the middleware. If you use a \textit{Subscriber}'s operations, the \textit{DataReader} will belong to that \textit{Subscriber}. So either way, the \textit{DataReader} belongs to a \textit{Subscriber}.

\textbf{Note:} Some operations cannot be used within a listener callback, see \textit{4.5.1 Restricted Operations in Listener Callbacks} on page 200.

\textbf{Table 8.3 DataReader Operations}

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Configuring the DataReader</td>
<td>enable</td>
<td>Enables the \textit{DataReader}.</td>
<td>4.1.2 Enabling DDS Entities on page 168</td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Compares two \textit{DataReader}'s QoS structures for equality.</td>
<td>8.3.9.2 Comparing QoS Values on page 548</td>
</tr>
<tr>
<td></td>
<td>get_qos</td>
<td>Gets the QoS.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_qos</td>
<td>Modifies the QoS.</td>
<td>8.3.9 Setting DataReader QosPolicies on page 543</td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>Modifies the QoS based on a QoS profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_listener</td>
<td>Gets the currently installed \textit{Listener}.</td>
<td>8.3.4 Setting Up DataReaderListeners on page 512</td>
</tr>
<tr>
<td></td>
<td>set_listener</td>
<td>Replaces the \textit{Listener}.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 8.3 DataReader Operations

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing DDS Data Samples with &quot;Read&quot;</td>
<td>read</td>
<td>Reads (copies) a collection of DDS data samples from the DataReader.</td>
<td>8.4.3 Accessing DDS Data Samples with Read or Take on page 555</td>
</tr>
<tr>
<td></td>
<td>read_instance</td>
<td>Identical to read, but all DDS samples returned belong to a single instance, which you specify as a parameter.</td>
<td>8.4.3.4 read_instance and take_instance on page 559</td>
</tr>
<tr>
<td></td>
<td>read_instance_w_condition</td>
<td>Identical to read_instance, but all DDS samples returned belong to a single instance and satisfy a specific ReadCondition.</td>
<td>8.4.3.7 read_instance_w_condition and take_instance_w_condition on page 561</td>
</tr>
<tr>
<td></td>
<td>read_next_instance</td>
<td>Similar to read_instance, but the actual instance is not directly specified as a parameter. Instead, the DDS samples will all belong to instance ordered after the one previously read.</td>
<td>8.4.3.5 read_next_instance and take_next_instance on page 559</td>
</tr>
<tr>
<td></td>
<td>read_next_instance_w_condition</td>
<td>Accesses a collection of DDS data samples of the next instance that match a specific set of ReadConditions, from the DataReader.</td>
<td>8.4.3.8 read_next_instance_w_condition and take_next_instance_w_condition on page 562</td>
</tr>
<tr>
<td></td>
<td>read_next_sample</td>
<td>Reads the next not-previously-accessed data value from the DataReader.</td>
<td>8.4.3.3 read_next_sample and take_next_sample on page 558</td>
</tr>
<tr>
<td></td>
<td>read_w_condition</td>
<td>Accesses a collection of DDS data samples from the DataReader that match specific ReadCondition criteria.</td>
<td>8.4.3.6 read_w_condition and take_w_condition on page 561</td>
</tr>
</tbody>
</table>
### Table 8.3 DataReader Operations

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accessing DDS Data Samples with “Take”</strong>&lt;br&gt; (Use FooData-Reader, see 8.4.3 Accessing DDS Data Samples with Read or Take on page 555)</td>
<td>take</td>
<td>Like read, but the DDS samples are removed from the <code>DataReader</code>'s receive queue.</td>
<td>8.4.3 Accessing DDS Data Samples with Read or Take on page 555</td>
</tr>
<tr>
<td></td>
<td>take_instance</td>
<td>Identical to take, but all DDS samples returned belong to a single instance, which you specify as a parameter.</td>
<td>8.4.3.4 <code>read_instance</code> and <code>take_instance</code> on page 559</td>
</tr>
<tr>
<td></td>
<td>take_instance_w_condition</td>
<td>Identical to <code>take_instance</code>, but all DDS samples returned belong to a single instance and satisfy a specific <code>ReadCondition</code>.</td>
<td>8.4.3.7 <code>read_instance.w_condition</code> and <code>take_instance.w_condition</code> on page 561</td>
</tr>
<tr>
<td></td>
<td>take_next_instance</td>
<td>Like <code>read_next_instance</code>, but the DDS samples are removed from the <code>DataReader</code>'s receive queue.</td>
<td>8.4.3.5 <code>read_next_instance</code> and <code>take_next_instance</code> on page 559</td>
</tr>
<tr>
<td></td>
<td>take_next_instance_w_condition</td>
<td>Accesses (and removes) a collection of DDS data samples of the next instance that match a specific set of <code>ReadConditions</code>, from the <code>DataReader</code>.</td>
<td>8.4.3.8 <code>read_next_instance.w_condition</code> and <code>take_next_instance.w_condition</code> on page 562</td>
</tr>
<tr>
<td></td>
<td>take_next_sample</td>
<td>Like <code>read_next_sample</code>, but the DDS samples are removed from the <code>DataReader</code>'s receive queue.</td>
<td>8.4.3.3 <code>read_next_sample</code> and <code>take_next_sample</code> on page 558</td>
</tr>
<tr>
<td><strong>Working with DDS Data Samples and FooData-Reader</strong>&lt;br&gt; (Use FooData-Reader, see 8.4.3 Accessing DDS Data Samples with Read or Take on page 555)</td>
<td>take_w_condition</td>
<td>Accesses (and removes) a collection of DDS data samples from the <code>DataReader</code> that match specific <code>ReadCondition</code> criteria.</td>
<td>8.4.3.6 <code>read_w_condition</code> and <code>take_w_condition</code> on page 561</td>
</tr>
<tr>
<td></td>
<td>narrow</td>
<td>A type-safe way to cast a pointer. This takes a DDSDataReader pointer and ‘narrows’ it to a ‘FooDataReader’ where ‘Foo’ is the related data type.</td>
<td>8.4.1 Using a Type-Specific DataReader (FooDataReader) on page 553</td>
</tr>
<tr>
<td></td>
<td>return_loan</td>
<td>Returns buffers loaned in a previous read or take call.</td>
<td>8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553</td>
</tr>
<tr>
<td></td>
<td>get_key_value</td>
<td>Gets the key for an instance handle.</td>
<td>8.3.10.5 Getting the Key Value for an Instance on page 553</td>
</tr>
<tr>
<td></td>
<td>lookup_instance</td>
<td>Gets the instance handle that corresponds to an instance key.</td>
<td>8.3.10.4 Looking Up an Instance Handle on page 552</td>
</tr>
<tr>
<td><strong>Acknowledging DDS Samples</strong></td>
<td>acknowledge_all</td>
<td>Acknowledge all previously accessed DDS samples.</td>
<td>8.4.4 Acknowledging DDS Samples on page 563</td>
</tr>
<tr>
<td></td>
<td>acknowledge_sample</td>
<td>Acknowledge a single DDS sample.</td>
<td>8.4.4 Acknowledging DDS Samples on page 563</td>
</tr>
</tbody>
</table>
### Table 8.3 DataReader Operations

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Checking Status</td>
<td><code>get_liveliness_changed_status</code></td>
<td>Gets LIVELINESS_CHANGED_STATUS status.</td>
<td>8.3.7 Statuses for DataReaders on page 516</td>
</tr>
<tr>
<td></td>
<td><code>get_requested_deadline_missed_status</code></td>
<td>Gets REQUESTED_DEADLINE_MISSED_STATUS status.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>get_requested_incompatible_qos_status</code></td>
<td>Gets REQUESTED_INCOMPATIBLE_QOS_STATUS status.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>get_sample_lost_status</code></td>
<td>Gets SAMPLE_LOST_STATUS status.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>get_sample_rejected_status</code></td>
<td>Gets SAMPLE_REJECTED_STATUS status.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>get_subscription_matched_status</code></td>
<td>Gets SUBSCRIPTION_MATCHED_STATUS status.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>get_status_changes</code></td>
<td>Gets a list of statuses that changed since last time the application read the status or the listeners were called.</td>
<td>4.1.4 Getting Status and Status Changes on page 171</td>
</tr>
<tr>
<td></td>
<td><code>get_datareader_cache_status</code></td>
<td>Gets DATA_READER_CACHE_STATUS status.</td>
<td>8.3.5 Checking DataReader Status and StatusConditions on page 514</td>
</tr>
<tr>
<td></td>
<td><code>get_datareader_protocol_status</code></td>
<td>Gets DATA_READER_PROTOCOL_STATUS status.</td>
<td>8.3.7 Statuses for DataReaders on page 516</td>
</tr>
<tr>
<td></td>
<td><code>get_matched_publication_datareader_protocol_status</code></td>
<td>Get the protocol status for this DataReader, per matched publication identified by the publication_handle.</td>
<td></td>
</tr>
</tbody>
</table>
## Table 8.3 DataReader Operations

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Navigating Relationships</td>
<td>get_instance_handle</td>
<td>Returns the DDS_InstanceHandle_t associated with the Entity.</td>
<td>4.1.3 Getting an Entity's Instance Handle on page 171</td>
</tr>
<tr>
<td></td>
<td>get_matched_publication_data</td>
<td>Gets information on a publication with a matching Topic and compatible QoS.</td>
<td>8.3.10.1 Finding Matching Publications on page 551</td>
</tr>
<tr>
<td></td>
<td>get_matched_publications</td>
<td>Gets a list of publications that have a matching Topic and compatible QoS. These are the publications currently associated with the DataReader.</td>
<td>8.3.10.2 Finding the Matching Publication’s ParticipantBuiltinTopicData on page 552</td>
</tr>
<tr>
<td></td>
<td>get_subscriber</td>
<td>Gets the Subscriber that created the DataReader.</td>
<td>8.3.10.3 Finding a DataReader’s Related Entities on page 552</td>
</tr>
<tr>
<td></td>
<td>get_topicdescription</td>
<td>Gets the Topic associated with the DataReader.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>is_matched_publication_alive</td>
<td>Enables you to query whether the matched DataWriter (using the instance handle returned by get_matched_publications) is alive. get_matched_publications returns all matching DataWriters, including those that are not alive. This operation enables you to see which matching DataWriters are alive.</td>
<td>8.3.10.1 Finding Matching Publications on page 551</td>
</tr>
<tr>
<td>Working with Conditions</td>
<td>create_query-condition</td>
<td>Creates a QueryCondition.</td>
<td>4.6.7 ReadConditions and QueryConditions on page 209</td>
</tr>
<tr>
<td></td>
<td>create_read-condition</td>
<td>Creates a ReadCondition.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete_read-condition</td>
<td>Deletes a ReadCondition/QueryCondition attached to the DataReader.</td>
<td>8.3.3.1 Deleting Contained ReadConditions on page 512</td>
</tr>
<tr>
<td></td>
<td>delete_contained_entities</td>
<td>Deletes all the ReadConditions/QueryConditions that were created by means of the &quot;create&quot; operations on the DataReader.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_status-condition</td>
<td>Gets the StatusCondition associated with the Entity.</td>
<td>4.6.8 StatusConditions on page 212</td>
</tr>
<tr>
<td></td>
<td>create_read-condition_w_params</td>
<td>Creates a ReadCondition with parameters.</td>
<td>4.6.7 ReadConditions and QueryConditions on page 209</td>
</tr>
<tr>
<td></td>
<td>create_query-condition_w_params</td>
<td>Creates a QueryCondition with parameters.</td>
<td>4.6.7 ReadConditions and QueryConditions on page 209</td>
</tr>
</tbody>
</table>
### 8.3.1 Creating DataReaders

Before you can create a `DataReader`, you need a `DomainParticipant` and a `Topic`.

*DataReaders* are created by calling `create_datareader()` or `create_datareader_with_profile()`—these operations exist for `DomainParticipants` and `Subscribers`. If you use the `DomainParticipant` to create a `DataReader`, it will belong to the implicit `Subscriber` described in 8.2.1 Creating Subscribers Explicitly vs. Implicitly on page 490. If you use a `Subscriber`’s operations to create a `DataReader`, it will belong to that `Subscriber`.

A QoS profile is a way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see Configuring QoS with XML (Chapter 19 on page 848).

Note: In the Modern C++ API, *DataReaders* provide constructors whose first argument is a `Subscriber`. The only required arguments are the subscriber and the topic.

```cpp
DDSDataReader* create_datareader(
    DDSTopicDescription *topic,
    const DDS_DataReaderQos &qos,
    DDSDataReaderListener *listener,
    DDS_StatusMask mask);

DDSDataReader * create_datareader_with_profile (  
    DDSTopicDescription * topic,
    const char * library_name,
    const char * profile_name,
    DDSDataReaderListener * listener,
    DDS_StatusMask mask)
```

Where:

- `topic` The *Topic* to which the *DataReader* is subscribing. This must have been previously created by the same `DomainParticipant`. 

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Working with Topic Queries</td>
<td><code>create_topic_query</code></td>
<td>Creates a TopicQuery. The returned TopicQuery will have been issued if the <code>DataReader</code> is enabled. Otherwise, the TopicQuery will be issued once the DataReader is enabled.</td>
<td>Topic Queries (Chapter 24 on page 977)</td>
</tr>
<tr>
<td></td>
<td><code>delete_topic_query</code></td>
<td>Deletes an active TopicQuery. After deleting a TopicQuery, new DataWriters won't discover it and existing DataWriters currently publishing cached samples may stop before delivering all of them.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>lookup_topic_query</code></td>
<td>Retrieves the TopicQuery that corresponds to the input GUID. To get the GUID associated with a TopicQuery, use the TopicQuery’s <code>get_guid()</code>.</td>
<td></td>
</tr>
<tr>
<td>Waiting for Historical Data</td>
<td><code>wait_for_historical_data</code></td>
<td>Waits until all &quot;historical&quot; (previously sent) data is received. Only valid for Reliable DataReaders with non-VOLATILE DURABILITY.</td>
<td>8.3.6 Waiting for Historical Data on page 515</td>
</tr>
</tbody>
</table>
8.3.1 Creating DataReaders

qos

If you want the default QoS settings (described in the API Reference HTML documentation), use DDS_DATAREADER_QOS_DEFAULT for this parameter (see Figure 8.9: Creating a DataReader with Default QosPolicies below). If you want to customize any of the QosPolicies, supply a QoS structure (see 8.3.9 Setting DataReader QosPolicies on page 543).

Note: If you use DDS_DATAREADER_QOS_DEFAULT for the qos parameter, it is not safe to create the DataReader while another thread may be simultaneously calling the Subscriber’s set_default_ datareader_qos() operation.

listener

A DataReader’s Listener is where you define the callback routine that will be notified when new DDS data samples arrive. Connext DDS also uses this Listener to notify your application of specific events (status changes) that may occur with respect to the DataReader. For more information, see 8.3.4 Setting Up DataReaderListeners on the next page and 8.3.7 Statuses for DataReaders on page 516.

The listener parameter is optional; you may use NULL instead. In that case, the Subscriber’s Listener (or if that is NULL, the DomainParticipant’s Listener) will receive the notifications instead. See 8.3.4 Setting Up DataReaderListeners on the next page for more on DataReaderListeners.

mask

This bit mask indicates which status changes will cause the Listener to be invoked. The bits set in the mask must have corresponding callbacks implemented in the Listener. If you use NULL for the Listener, use DDS_STATUS_MASK_NONE for this parameter. If the Listener implements all callbacks, use DDS_STATUS_MASK_ALL. For information on statuses, see 4.4 Listeners on page 189.

library_name

A QoS Library is a named set of QoS profiles. See 19.3 QoS Profiles on page 850.

profile_name

A QoS profile groups a set of related QoS, usually one per entity. See 19.3 QoS Profiles on page 850.

After you create a DataReader, you can use it to retrieve received data. See 8.4 Using DataReaders to Access Data (Read & Take) on page 553.

Note: When a DataReader is created, only those transports already registered are available to the DataReader. The built-in transports are implicitly registered when (a) the DomainParticipant is enabled, (b) the first DataReader is created, or (c) you lookup a built-in DataReader, whichever happens first.

Figure 8.9: Creating a DataReader with Default QosPolicies below shows an example of how to create a DataReader with default QosPolicies.

Figure 8.9: Creating a DataReader with Default QosPolicies

```cpp
// MyReaderListener is user defined, extends DDSDataReaderListener
DDSDataReaderListener *reader_listener = new MyReaderListener();
DataReader* reader = subscriber->create_datareader(topic,
    DDS_DATAREADER_QOS_DEFAULT,
    reader_listener, DDS_STATUS_MASK_ALL);
if (reader == NULL) {
    // ... error
}
// narrow it into your specific data type
FooDataReader* foo_reader = FooDataReader::narrow(reader);
```

For more examples on how to create a DataReader, see 8.3.9.1 Configuring QoS Settings when theDataReader is Created on page 546.
8.3.2 Getting All DataReaders

To retrieve all the DataReaders created by the Subscriber, use the Subscriber’s `get_all_datareaders()` operation:

```c
DDS_ReturnCode_t get_all_datareaders(    DDS_Subscriber* self,    struct DDS_DataReaderSeq* readers);
```

In the Modern C++ API, use the freestanding function `rti::sub::find_datareaders()`.

8.3.3 Deleting DataReaders

(Note: in the Modern C++ API, Entities are automatically destroyed, see 4.1.1 Creating and Deleting DDS Entities on page 167)

To delete a DataReader:

Delete any ReadConditions and QueryConditions that were created with the DataReader. Use the DataReader’s `delete_readcondition()` operation to delete them one at a time, or use the `delete_contained_entities()` operation (8.3.3.1 Deleting Contained ReadConditions below) to delete them all at the same time.

```c
DDS_ReturnCode_t delete_readcondition (DDSReadCondition *condition)
```

Delete the DataReader by using the Subscriber’s `delete_datareader()` operation (8.2.3 Deleting Subscribers on page 492).

Note: A DataReader cannot be deleted within its own reader listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

To delete all of a Subscriber’s DataReaders, use the Subscriber’s `delete_contained_entities()` operation (see 8.2.3.1 Deleting Contained DataReaders on page 493).

8.3.3.1 Deleting Contained ReadConditions

The DataReader’s `delete_contained_entities()` operation deletes all the ReadConditions and QueryConditions (4.6.7 ReadConditions and QueryConditions on page 209) that were created by the DataReader.

```c
DDS_ReturnCode_t delete_contained_entities ()
```

After this operation returns successfully, the application may delete the DataReader (see 8.3.3 Deleting DataReaders above).

8.3.4 Setting Up DataReaderListeners

DataReaders may optionally have Listeners. A DataReaderListener is a collection of callback methods; these methods are invoked by Connext DDS when DDS data samples are received or when there are status changes for the DataReader.
8.3.4 Setting Up DataReaderListeners

Note: Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

If you do not implement a DataReaderListener, the associated Subscriber’s Listener is used instead. If that Subscriber does not have a Listener either, then the DomainParticipant’s Listener is used if one exists (see 8.2.6 Setting Up SubscriberListeners on page 500 and 9.3.6 Setting Up DomainParticipantListeners on page 620).

If you do not require asynchronous notification of data availability or status changes, you do not need to set a Listener for the DataReader. In that case, you will need to periodically call one of the read() or take() operations described in 8.4 Using DataReaders to Access Data (Read & Take) on page 553 to access the data that has been received.

Listeners are typically set up when the DataReader is created (see 8.3.1 Creating DataReaders on page 510). You can also set one up after creation by using the DataReader’s get_listener() and set_listener() operations. Connext DDS will invoke a DataReader’s Listener to report the status changes listed in Table 8.4 DataReaderListener Callbacks (if the Listener is set up to handle the particular status, see 8.3.4 Setting Up DataReaderListeners on the previous page).

Table 8.4 DataReaderListener Callbacks

<table>
<thead>
<tr>
<th>This DataReaderListener callback...</th>
<th>...is triggered by a change in this status:</th>
</tr>
</thead>
<tbody>
<tr>
<td>on_data_available()</td>
<td>8.3.7.1 DATAAVAILABLE Status on page 517</td>
</tr>
<tr>
<td>on_liveliness_changed()</td>
<td>8.3.7.4 LIVELINESS_CHANGED Status on page 524</td>
</tr>
<tr>
<td>on_requested_deadline_missed()</td>
<td>8.3.7.5 REQUESTED_DEADLINE_MISSED Status on page 526</td>
</tr>
<tr>
<td>on_requested_incompatible_qos()</td>
<td>8.3.7.6 REQUESTED_INCOMPATIBLE_QOS Status on page 526</td>
</tr>
<tr>
<td>on_sample_lost()</td>
<td>8.3.7.7 SAMPLE_LOST Status on page 527</td>
</tr>
<tr>
<td>on_sample_rejected()</td>
<td>8.3.7.8 SAMPLE_REJECTED Status on page 531</td>
</tr>
<tr>
<td>on_subscription_matched()</td>
<td>8.3.7.9 SUBSCRIPTION_M Matched Status on page 533</td>
</tr>
</tbody>
</table>

Note that the same callbacks can be implemented in the SubscriberListener or DomainParticipantListener instead. There is only one SubscriberListener callback that takes precedence over a DataReaderListener’s. An on_data_on_readers() callback in the SubscriberListener (or DomainParticipantListener) takes precedence over the on_data_available() callback of a DataReaderListener.

If the SubscriberListener implements an on_data_on_readers() callback, it will be invoked instead of the DataReaderListener’s on_data_available() callback when new data arrives. The on_data_on_readers() operation can in turn cause the on_data_available() method of the appropriate DataReaderListener to be invoked by calling the Subscriber’s notify_datareaders() operation. For more information on status and Listeners, see 4.4 Listeners on page 189.
8.3.5 Checking DataReader Status and StatusConditions

You can access individual communication status for a DataReader with the operations shown in Table 1 DataReader Status Operations.
8.3.6 Waiting for Historical Data

Table 1 DataReader Status Operations

<table>
<thead>
<tr>
<th>Use this operation...</th>
<th>...to retrieve this status:</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_datareader_cache_status</td>
<td>8.3.7.2 DATA_READER_CACHE_STATUS on page 517</td>
</tr>
<tr>
<td>get_datareader_protocol_status</td>
<td>8.3.7.3 DATA_READER_PROTOCOL_STATUS on page 520</td>
</tr>
<tr>
<td>get_matched_publication_datareader_protocol_status</td>
<td>8.3.7.4 LIVELINESS_CHANGED Status on page 524</td>
</tr>
<tr>
<td>get_liveliness_changed_status</td>
<td>8.3.7.5 REQUESTED_DEADLINE_MISSED Status on page 526</td>
</tr>
<tr>
<td>get_sample_lost_status</td>
<td>8.3.7.7 SAMPLE_LOST Status on page 527</td>
</tr>
<tr>
<td>get_sample_rejected_status</td>
<td>8.3.7.8 SAMPLE_REJECTED Status on page 531</td>
</tr>
<tr>
<td>get_requested_deadline_missed_status</td>
<td>8.3.7.9 SUBSCRIPTION_MATCHED Status on page 533</td>
</tr>
<tr>
<td>get_requested_incompatible_qos_status</td>
<td>All of the above</td>
</tr>
<tr>
<td>get_subscription_match_status</td>
<td>See 4.6.8 StatusConditions on page 212</td>
</tr>
</tbody>
</table>

These methods are useful in the event that no Listener callback is set to receive notifications of status changes. If a Listener is used, the callback will contain the new status information, in which case calling these methods is unlikely to be necessary.

The get_status_changes() operation provides a list of statuses that have changed since the last time the status changes were ‘reset.’ A status change is reset each time the application calls the corresponding get_*_status(), as well as each time Connext DDS returns from calling the Listener callback associated with that status.

For more on status, see 8.3.4 Setting Up DataReaderListeners on page 512, 8.3.7 Statuses for DataReaders on the next page, and 4.4 Listeners on page 189.

8.3.6 Waiting for Historical Data

The wait_for_historical_data() operation waits (blocks) until all "historical" data is received from matched DataWriters. "Historical" data means DDS samples that were written before the DataReader joined the DDS domain.

This operation is intended only for DataReaders that have:

- 7.5.9 DURABILITY QosPolicy on page 411 kind set to TRANSIENT_LOCAL (not VOLATILE)
- 7.5.21 RELIABILITY QosPolicy on page 447 kind set to RELIABLE
8.3.7 Statuses for DataReaders

Calling \texttt{wait\_for\_historical\_data()} on a non-reliable \textit{DataReader} will always return immediately, since \textit{Connext DDS} will never deliver historical data to non-reliable \textit{DataReaders}.

As soon as an application enables a non-VOLATILE \textit{DataReader}, it will start receiving both "historical" data as well as any new data written by matching \textit{DataWriters}. If you want the subscribing application to wait until all "historical" data is received, use this operation:

\begin{verbatim}
DDS_ReturnCode_t wait_for_historical_data (const DDS_Duration_t & max_wait)
\end{verbatim}

The \texttt{wait\_for\_historical\_data()} operation blocks the calling thread until either all "historical" data is received or the duration specified by the \texttt{max\_wait} parameter elapses, whichever happens first. A return value of OK indicates that all the "historical" data was received; a return value of TIMEOUT indicates that \texttt{max\_wait} elapsed before all the data was received.

\texttt{wait\_for\_historical\_data()} will return immediately if no \textit{DataWriters} have been discovered at the time the operation is called. Therefore it is advisable to make sure at least one \textit{DataWriter} has been discovered before calling this operation; one way to do this is to use \texttt{get\_subscription\_matched\_status()}, like this:

\begin{verbatim}
while (1) {
    DDS_SubscriptionMatchedStatus status;
    MyType_reader->get_subscription_matched_status(status);
    if (status.current_count > 0) { break; }
    NDDS::sleep(sleep_period);
}
\end{verbatim}

8.3.7 Statuses for DataReaders

There are several types of statuses available for a \textit{DataReader}. You can use the \texttt{get\_*\_status()} operations (8.3.5 Checking DataReader Status and StatusConditions on page 514) to access and reset them, use a \textit{DataReaderListener} (8.3.4 Setting Up DataReaderListeners on page 512) to listen for changes in their values (for those statuses that have Listeners), or use a \textit{StatusCondition} and a \textit{WaitSet} (4.6.8 StatusConditions on page 212) to wait for changes. Each status has an associated data structure and is described in more detail in the following sections.

- 8.3.7.1 \texttt{DATA\_AVAILABLE} Status on the next page
- 8.3.7.2 \texttt{DATA\_READER\_CACHE\_STATUS} Status on the next page
- 8.3.7.3 \texttt{DATA\_READER\_PROTOCOL\_STATUS} on page 520
- 8.3.7.4 \texttt{LIVELINESS\_CHANGED} Status on page 524
- 8.3.7.5 \texttt{REQUESTED\_DEADLINE\_MISSED} Status on page 526
- 8.3.7.6 \texttt{REQUESTED\_INCOMPATIBLE\_QOS} Status on page 526
- 8.3.7.7 \texttt{SAMPLE\_LOST} Status on page 527
- 8.3.7.8 \texttt{SAMPLE\_REJECTED} Status on page 531
- 8.3.7.9 \texttt{SUBSCRIPTION\_MATCHED} Status on page 533
8.3.7.1 DATA_AVAILABLE Status

This status indicates that new data is available for the DataReader. In most cases, this means that one new DDS sample has been received. However, there are situations in which more than one DDS samples for the DataReader may be received before the DATA_AVAILABLE status changes. For example, if the DataReader has the 7.5.9 DURABILITY QosPolicy on page 411 set to be non-VOLATILE, then the DataReader may receive a batch of old DDS data samples all at once. Or if data is being received reliably from DataWriters, Connext DDS may present several DDS samples of data simultaneously to the DataReader if they have been originally received out of order.

A change to this status also means that the DATA_ON_READERS status is changed for the DataReader’s Subscriber. This status is reset when you call read(), take(), or one of their variations.

Unlike most other statuses, this status (as well as DATA_ON_READERS for Subscribers) is a read communication status. See 8.2.9 Statuses for Subscribers for page 503 and 4.3.1 Types of Communication Status on page 183 for more information on read communication statuses.

The DataReaderListener’s on_data_available() callback is invoked when this status changes, unless the SubscriberListener (8.2.6 Setting Up SubscriberListeners on page 500) or DomainParticipantListener (9.3.6 Setting Up DomainParticipantListeners on page 620) has implemented an on_data_on_readers() callback. In that case, on_data_on_readers() will be invoked instead.

8.3.7.2 DATA_READER_CACHE_STATUS

This status keeps track of the number of DDS samples and instances in the reader's cache, including the number of samples that were dropped for different reasons. For information on the instance states described in the reader's cache, such as "alive," "no_writers," and "disposed," see 6.1 Instance States on page 258.

This status does not have an associated Listener. You can access this status by calling the DataReader’s get_datareader_cache_status() operation, which will return the status structure described in Table 8.5 DDS_DataReaderCacheStatus.

Table 8.5 DDS_DataReaderCacheStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>sample_count_peak</td>
<td>Highest number of DDS samples in the DataReader's queue over the lifetime of the DataReader.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sample_count</td>
<td>Current number of DDS samples in the DataReader's queue. Includes DDS samples that may not yet be available to be read or taken by the user due to DDS samples being received out of order or settings in the 7.4.6 PRESENTATION QosPolicy on page 362.</td>
</tr>
</tbody>
</table>
### Table 8.5 DDS_DataReaderCacheStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>writer_removed_batch_sample_dropped_sample_count</td>
<td>The number of batched samples received by the DataReader that were marked as removed by the DataWriter. Examples of removed samples in a batch are samples that were replaced due to KEEP_LAST_HISTORY_QOS on the DataWriter or samples that outlived the DataWriter’s LIFESPAN QosPolicy on page 425. By default, any sample marked as removed from a batch is dropped, unless you set the DDS_data_reader.accept_writer_removed_batch_samples property in the DDS Extension to TRUE. (By default, it is set to FALSE.) Note: Historical data with removed batch samples written before the DataReader joined the DDS domain are also included in the count.</td>
</tr>
</tbody>
</table>
| DDS_LongLong| old_source_timestamp_dropped_sample_count            | The number of samples dropped as a result of receiving a sample older than the last one, using DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS. When the DataReader is using DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS:  
- If the DataReader receives a sample for an instance with a source timestamp that is older than the last source timestamp received for the instance, the sample is dropped and included in this count.  
- If the DataReader receives a sample for an instance with a source timestamp that is equal to the last source timestamp received for the instance and the writer has a higher virtual GUID, the sample is dropped and included in this count. |
| DDS_LongLong| tolerance_source_timestamp_dropped_sample_count      | The number of samples dropped as a result of receiving a sample in the future, using DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS. When the DataReader is using DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS, the DataReader will accept a sample only if the source timestamp is no farther in the future from the reception timestamp than the source_timestamp_tolerance. Otherwise, the sample is dropped and included in this count. |
| DDS_LongLong| ownership_dropped_sample_count                       | The number of samples dropped as a result of receiving a sample from a DataWriter with a lower strength, using Exclusive Ownership. When using Exclusive Ownership, the DataReader receives data from multiple DataWriters. Each instance can only be owned by one DataReader. If other DataWriters write samples belonging to this instance, the samples will be dropped. |
| DDS_LongLong| content_filter_dropped_sample_count                  | The number of samples filtered by the DataReader due to ContentFilteredTopics. When using a content filter on the DataReader side, if the sample received by the DataReader does not pass the filter, it will be dropped. |
| DDS_LongLong| time_based_filter_dropped_sample_count               | The number of samples filtered by the DataReader due to the 8.6.4 TIME_BASED_FILTER QosPolicy. When using the 8.6.4 TIME_BASED_FILTER QosPolicy on page 588 on the DataReader side, if the sample received by the DataReader does not pass the minimum_separation filter, it will be dropped. |
Table 8.5 DDS_DataReaderCacheStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>expired_dropped_sample_count</td>
<td>The number of samples expired by the DataReader due to the DDS_LifespanQosPolicy on page 425 or the autpurge sample delays in the 8.6.3 READER_DATA_LIFECYCLE QoS Policy on page 586:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_LifespanQosPolicy: When a sample expires due to the DDS_LifespanQosPolicy, the data is removed from the DataReader caches. This sample will be considered dropped if its DDS_SampleStateKind is DDS_NOT_READ_SAMPLE_STATE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_ReaderDataLifecycleQosPolicy::autpurge_nowriter_samples_delay: When a sample expires due to the autpurge_nowriter_samples_delay, this sample will be considered dropped if its DDS_SampleStateKind is DDS_NOT_READ_SAMPLE_STATE.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_ReaderDataLifecycleQosPolicy::autpurge_disposed_samples_delay: When a sample expires due to the autpurge_disposed_samples_delay, this sample will be considered dropped if its DDS_SampleStateKind is DDS_NOT_READ_SAMPLE_STATE.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>virtual_duplicate_dropped_sample_count</td>
<td>The number of virtual duplicate samples dropped by the DataReader. A sample is a virtual duplicate if it has the same identity (Virtual Writer GUID and Virtual Sequence Number) as a previously received sample.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When two DataWriters with the same logical data source publish a sample with the same sequence_number, one sample will be dropped and the other will be received by the DataReader. This can happen when multiple writers are writing on behalf of the same original DataWriter; for example, in systems with redundant RTI Routing Service applications or when a DataReader is receiving samples both directly from the original DataWriter and from an instance of RTI Persistence Service.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>replaced_dropped_sample_count</td>
<td>The number of samples replaced by the DataReader due to DDS_KEEP_LAST_HISTORY_QOS replacement in the 7.5.12 HISTORY QosPolicy on page 420.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When the number of samples for an instance in the queue reaches the depth value in the HISTORY QosPolicy, a new sample for the instance will replace the oldest sample for the instance in the queue. The new sample will be accepted, and the old sample will be dropped. This counter will only be updated if the replaced sample’s DDS_SampleStateKind is DDS_NOT_READ_SAMPLE_STATE.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>total_samples_dropped_by_instance_replacement</td>
<td>Number of samples of the state NOT_READ_SAMPLE_STATE that were dropped when removing an instance due to instance replacement via the instance_replacement field in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>alive_instance_count</td>
<td>Number of instances currently in the DataReader’s queue that have an instance_state of ALIVE.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>alive_instance_count_peak</td>
<td>Highest number of ALIVE instances in the DataReader’s queue over the lifetime of the DataReader.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>no_writers_instance_count</td>
<td>Number of instances in the DataReader’s queue that have an instance_state of NOT_ALIVE_NO_WRITERS.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>no_writers_instance_count_peak</td>
<td>Highest number of NOT_ALIVE_NO_WRITERS instances in the DataReader’s queue over the lifetime of the DataReader.</td>
</tr>
</tbody>
</table>
Table 8.5 DDS_DataReaderCacheStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>disposed_instance_count</td>
<td>Number of instances in the DataReader's queue that have an instance state of NOT_ALIVE_. DISPOSED.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>disposed_instance_count_peak</td>
<td>Highest number of NOT_ALIVE.DISPOSED instances in the DataReader's queue over the lifetime of the DataReader.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>detached_instance_count</td>
<td>Number of detached instances—which contain only the minimum instance state—currently being maintained in the DataReader's queue. If keep_minimum_state_for_instances in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576 is true (by default, it is), the DataReader will keep up to max_total_instances (in the DATA_READERRESOURCE_LIMITS QosPolicy) of detached instances in its queue. See 8.3.8.6.1 Active State and Minimum State on page 540 for more information.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>detached_instance_count_peak</td>
<td>Highest number of detached instances in the DataReader's queue over the lifetime of the DataReader.</td>
</tr>
</tbody>
</table>

8.3.7.3 DATA_READER_PROTOCOL_STATUS

The status of a DataReader’s internal protocol related metrics (such as the number of DDS samples received, filtered, rejected) and the status of wire protocol traffic. The structure for this status appears in Table 8.6 DDS_DataReaderProtocolStatus.

This status does not have an associated Listener. You can access this status by calling the following operations on the DataReader (which return the status structure described in Table 8.6 DDS_DataReaderProtocolStatus):

get_datareader_protocol_status() returns the sum of the protocol status for all the matched publications for the DataReader.

get_matched_publication_datareader_protocol_status() returns the protocol status of a particular matched publication, identified by a publication_handle.

The get_*_status() operations also reset the related status so it is no longer considered “changed.”

Note: Status/data for a matched publication is kept even if the DataWriter is not alive (that is, has lost liveliness based on the 7.5.15 Liveliness QosPolicy on page 427). The status/data will be removed only if the DataWriter is gone: that is, the DataWriter is destroyed and this change is propagated through a discovery update, or the DataWriter's DomainParticipant is gone (either gracefully or its liveliness expired and Connext DDS is configured to purge not-alive participants). Once a matched DataWriter is gone, its status is deleted. If you try to get the status/data for a matched publication that is gone, the 'get status' or 'get data' call will return an error.

The DataReader's protocol status includes information about DATA_FRAG messages (sample fragments) if you are using DDS-level fragmentation. See 23.3 Large Data Fragmentation on page 970 for more information.
## 8.3.7 Statuses for DataReaders

### Table 8.6 DDS_DataReaderProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>received_sample_count</td>
<td>The number of samples received by a DataReader. <strong>Note:</strong> When data is fragmented, this count is updated when all of the fragments required to reassemble a sample are received, not when individual fragments are received. The fragment count is tracked in the received_fragment_count.</td>
</tr>
<tr>
<td></td>
<td>received_sample_count_change</td>
<td>Change in the received_sample_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_sample_bytes</td>
<td>The number of bytes received by a DataReader. <strong>Note:</strong> When data is fragmented, this statistic is updated upon the receipt of each fragment, not when a sample is reassembled.</td>
</tr>
<tr>
<td></td>
<td>received_sample_bytes_change</td>
<td>Change in received_sample_bytes since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>duplicate_sample_count</td>
<td>The number of DDS samples received from a DataWriter, not for the first time, by this DataReader.</td>
</tr>
<tr>
<td></td>
<td>duplicate_sample_count_change</td>
<td>Change in duplicate_sample_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>duplicate_sample_bytes</td>
<td>The number of bytes of DDS samples received from a DataWriter, not for the first time, by this DataReader.</td>
</tr>
<tr>
<td></td>
<td>duplicate_sample_bytes_change</td>
<td>Change in the duplicate_sample_bytes since the last time the status was read.</td>
</tr>
</tbody>
</table>
### Table 8.6 DDS_DataReaderProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>filtered_sample_count</td>
<td>The number of DDS samples filtered by this DataReader due to ContentFilteredTopics or Time-Based Filter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>filtered_sample_count_change</td>
<td>Change in the filtered_sample_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>filtered_sample_bytes</td>
<td>The number of bytes of DDS samples filtered by this DataReader due to ContentFilteredTopics or Time-Based Filter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>filtered_sample_bytes_change</td>
<td>Change in the filtered_sample_bytes since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_heartbeat_count</td>
<td>The number of Heartbeats received from a DataWriter by this DataReader.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_heartbeat_count_change</td>
<td>Change in the received_heartbeat_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_heartbeat_bytes</td>
<td>The number of bytes of Heartbeats received from a DataWriter by this DataReader.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_heartbeat_bytes_change</td>
<td>Change in the received_heartbeat_bytes since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_ack_count</td>
<td>The number of ACKs sent from this DataReader to a matching DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_ack_count_change</td>
<td>Change in the sent_ack_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_ack_bytes</td>
<td>The number of bytes of ACKs sent from this DataReader to a matching DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_ack_bytes_change</td>
<td>Change in the sent_ack_bytes since the last time the status was read.</td>
</tr>
</tbody>
</table>
## 8.3.7 Statuses for DataReaders

### Table 8.6 DDS_DataReaderProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>sent_nack_count</td>
<td>The number of NACKs sent from this DataReader to a matching DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_nack_count_change</td>
<td>Change in the sent_nack_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_nack_bytes</td>
<td>The number of bytes of NACKs sent from this DataReader to a matching DataWriter.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_nack_bytes_change</td>
<td>Change in the sent_nack_bytes since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_gap_count</td>
<td>The number of GAPs received from a DataWriter to this DataReader.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_gap_count_change</td>
<td>Change in the received_gap_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_gap_bytes</td>
<td>The number of bytes of GAPs received from a DataWriter to this DataReader.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_gap_bytes_change</td>
<td>Change in the received_gap_bytes since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>rejected_sample_count</td>
<td>The number of times a sample is rejected because it cannot be accepted by a reliable DataReader. Samples rejected by a reliable DataReader will be NACKed, and they will have to be resent by the DataWriter if they are still available in the DataWriter queue. Note: This count is a subset of the total_count in the 8.3.7.8 SAMPLE_REJECTED Status on page 531. The total_count in the SAMPLE_REJECTED status includes both protocol-related rejections, that trigger a repair or resend (the rejected_sample_count described here), and the rejections described in the 8.3.7.8 SAMPLE_REJECTED Status on page 531. For example, the DDS_REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT in the SAMPLE_REJECTED status is not part of the rejected_sample_count because it does not trigger a repair or resend.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>rejected_sample_count_change</td>
<td>Change in the rejected_sample_count since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>out_of_range_rejected_sample_count</td>
<td>The number of samples dropped by the DataReader due to the receive window being full and the sample received out-of-order. When using reliable 7.5.21 RELIABILITY QosPolicy on page 447, if the DataReader receives samples out-of-range, they are stored internally until the missing samples are received. The number of out-of-order samples that the DataReader can keep is set by the receive_window_size in the Table 8.19 DDS_RtpsReliableReaderProtocol_t on page 573. When the receive window is full, any out-of-order sample received will be dropped and included in this count (but not in the SampleRejectedStatus).</td>
</tr>
</tbody>
</table>
Table 8.6 DDS_DataReaderProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_SequenceNumber</td>
<td>first_available_sample_sequence_number</td>
<td>Sequence number of the first available DDS sample in a matched DataWriter’s reliability queue. Applicable only when retrieving matched DataWriter statuses.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>last_available_sample_sequence_number</td>
<td>Sequence number of the last available DDS sample in a matched DataWriter’s reliability queue. Applicable only when retrieving matched DataWriter statuses.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>last_committed_sample_sequence_number</td>
<td>Sequence number of the last committed DDS sample (i.e. available to be read or taken) in a matched DataWriter’s reliability queue. Applicable only when retrieving matched DataWriter statuses. For best-effort DataReaders, this is the sequence number of the latest DDS sample received. For reliable DataReaders, this is the sequence number of the latest DDS sample that is available to be read or taken from the DataReader’s queue.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>uncommitted_sample_count</td>
<td>Number of received DDS samples that are not yet available to be read or taken due to being received out of order. Applicable only when retrieving matched DataWriter statuses.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>received_fragment_count</td>
<td>The number of fragments (DATA_FRAG messages) that have been received by this DataReader. This count is incremented upon the receipt of each DATA_FRAG message. Fragments from duplicate samples do not count towards this number. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>dropped_fragment_count</td>
<td>The number of DATA_FRAG messages that have been dropped by the DataReader. This count does not include malformed fragments. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>reassembled_sample_count</td>
<td>The number of samples that have been reassembled by the DataReader. This statistic is incremented when all of the fragments that are required to reassemble an entire sample have been received. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_nack_fragment_count</td>
<td>The number of NACK_FRAG RTPS messages that have been sent from the DataReader to a DataWriter. NACK_FRAG RTPS messages are sent when large data is used in conjunction with reliable communication. They have the same properties as NACK messages, but instead of applying to samples, they apply to fragments. Applicable only when data is fragmented.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>sent_nack_fragment_bytes</td>
<td>The number of NACK_FRAG message bytes that have been sent from the DataReader to a DataWriter. NACK_FRAG RTPS messages are sent when large data is used in conjunction with reliable communication. They have the same properties as NACK messages, but instead of applying to samples, they apply to fragments. Applicable only when data is fragmented.</td>
</tr>
</tbody>
</table>

8.3.7.4 LIVELINESS_CHANGED Status

This status indicates that the liveliness of one or more matched Data Writers has changed (i.e., one or more Data Writers has become alive or not alive). The mechanics of determining liveliness between a Data Writer and a Data Reader is specified in their 7.5.15 LIVELINESS QosPolicy on page 427.

The structure for this status appears in Table 8.7 DDS_LivelinessChangedStatus.
### Table 8.7 DDS_LivelinessChangedStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>alive_count</td>
<td>Number of matched DataWriters that are currently alive.</td>
</tr>
<tr>
<td></td>
<td>not_alive_count</td>
<td>Number of matched DataWriters that are not currently alive.</td>
</tr>
<tr>
<td></td>
<td>alive_count_change</td>
<td>The change in the alive_count since the last time the Listener was called or the status was read.</td>
</tr>
<tr>
<td></td>
<td>not_alive_count_change</td>
<td>The change in the not_alive_count since the last time the Listener was called or the status was read. Note that a positive not_alive_count_change means one of the following:</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>last_publication_handle</td>
<td>This InstanceHandle can be used to look up which remote DataWriter was the last to cause this DataReader's status to change, using the DataReader's get_matched_publication_data() method. It's possible that the DataWriter has been purged from the discovery database. If so, get_matched_publication_data() will not be able to return information about the DataWriter. In this case, the only way to get information about the lost DataWriter is if you cached the information previously.</td>
</tr>
</tbody>
</table>

The **DataReaderListener's on_liveliness_changed()** callback may be called for the following reasons:

- The liveliness of any DataWriter matching this DataReader (as defined by the 7.5.15 LIVELINESS QosPolicy on page 427) is lost.
- A DataWriter's liveliness is recovered after being lost.
- A new matching DataWriter has been discovered.
- A QoS Policy has changed such that a DataWriter that matched this DataReader before no longer matches (such as a change to the PartitionQosPolicy). In this case, Connext DDS will no longer keep track of the DataWriter's liveliness. Furthermore:
  - If the DataWriter was alive when it and the DataReader stopped matching: alive_count will decrease (since there's one less matching alive DataWriter) and not_alive_count will remain the same (since the DataWriter is still alive).
  - If the DataWriter was not alive when it and the DataReader stopped matching: alive_count will remain the same (since the matching DataWriter was not alive) and not_alive_count will decrease (since there's one less not-alive matching DataWriter).

**Note:** There are several ways that a DataWriter and DataReader can become incompatible after the DataWriter has lost liveliness. For example, when the 7.5.15 LIVELINESS QosPolicy on page 427 kind is set to MANUAL_BY_PARTICIPANT_LIVELINESS_QOS, it is possible that the DataWriter has not asserted its liveliness in a timely manner, and
then a QoS change occurs on the DataWriter or DataReader that makes the entities incompatible.

- A QoS Policy (such as the PartitionQosPolicy) has changed such that a DataWriter that was unmatched with the DataReader now matches.

You can also retrieve the value by calling the DataReader’s `get_liveliness_changed_status()` operation; this will also reset the status so it is no longer considered “changed.”

This status is reciprocal to the 7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) on page 307 for a DataWriter.

### 8.3.7.5 REQUESTED_DEADLINE_MISSED Status

This status indicates that the DataReader did not receive a new DDS sample for an data-instance within the time period set in the DataReader’s 7.5.7 DEADLINE QosPolicy on page 405. For non-keyed Topics, this simply means that the DataReader did not receive data within the DEADLINE period. For keyed Topics, this means that for one of the data-instances that the DataReader was receiving, it has not received a new DDS sample within the DEADLINE period. For more information about keys and instances, see 2.4 DDS Samples, Instances, and Keys on page 18.

The structure for this status appears in Table 8.8 DDS_RequestedDeadlineMissedStatus.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative number of times that the deadline was violated for any instance read by the DataReader.</td>
</tr>
<tr>
<td></td>
<td>total_count_change</td>
<td>The change in total_count since the last time the Listener was called or the status was read.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>last_instance_handle</td>
<td>Handle to the last data-instance in the DataReader for which a requested deadline was missed.</td>
</tr>
</tbody>
</table>

The DataReaderListener’s on_requested_deadline_missed() callback is invoked when this status changes. You can also retrieve the value by calling the DataReader’s `get_requested_deadline_missed_status()` operation; this will also reset the status so it is no longer considered “changed.”

### 8.3.7.6 REQUESTED_INCOMPATIBLE_QOS Status

A change to this status indicates that the DataReader discovered a DataWriter for the same Topic, but that DataReader had requested QoS settings incompatible with this DataWriter’s offered QoS.

The structure for this status appears in Table 8.9 DDS_RequestedIncompatibleQosStatus.
8.3.7 Statuses for DataReaders

Table 8.9 DDS_RequestedIncompatibleQosStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative number of times the DataReader discovered a DataWriter for the same Topic with an offered QoS that is incompatible with that requested by the DataReader.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>total_count_change</td>
<td>The change in total_count since the last time the Listener was called or the status was read.</td>
</tr>
<tr>
<td>DDS_QosPolicyId_t</td>
<td>last_policy_id</td>
<td>The ID of the QosPolicy that was found to be incompatible the last time an incompatibility was detected. (Note: if there are multiple incompatible policies, only one of them is reported here.)</td>
</tr>
<tr>
<td>DDS_QosPolicyCountSeq</td>
<td>policies</td>
<td>A list containing—for each policy—the total number of times that the DataReader discovered a DataWriter for the same Topic with a offered QoS that is incompatible with that requested by the DataReader.</td>
</tr>
</tbody>
</table>

The DataReaderListener’s on_requested_incompatible_qos() callback is invoked when this status changes. You can also retrieve the value by calling the DataReader’s get_requested_incompatible_qos_status() operation; this will also reset the status so it is no longer considered “changed.”

8.3.7.7 SAMPLE_LOST Status

This status indicates that one or more DDS samples written by a matched DataWriter have failed to be received and will never be received.

Some samples written by a DataWriter to its matching DataReaders may never be received by the DataReaders. This can happen because something went wrong while trying to add the sample to the DataReader’s queue, like a decryption or deserialization error, or because the sample was removed from the DataWriter’s queue before it was received by the DataReaders. A sample can be removed from the DataWriter’s queue before it is delivered to matching DataReaders for a number of reasons, including that DataWriters are limited in the number of published DDS data samples that they can store, so that if a DataWriter continues to publish DDS data samples, new data may overwrite old data that has not yet been received by the DataReader. The DDS samples that are overwritten can never be resent to the DataReader and thus are considered to be lost. DataWriters may also set the 7.5.14 LIFESPAN QoS Policy on page 425, and samples that expire due to lifespan may also be reported as lost by a DataReader that has not received those samples.

The lost status applies to reliable and best-effort DataReaders, see the 7.5.21 RELIABILITY QosPolicy on page 447. By reporting a sample as lost, the DataReader has declared that the sample will never be received, and will therefore not NACK it. It cannot be repaired by a DataWriter or resent to the DataReader.

Before a sample is received by a DataReader it may also be reported as rejected or dropped. (See 8.3.7.8 SAMPLE_REJECTED Status on page 531 and 8.3.7.2 DATA_READER_CACHE_STATUS on page 517.)
The structure for the *lost* status appears in Table 8.10 DDS_SampleLostStatus.

**Table 8.10 DDS_SampleLostStatus**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative count of all the DDS samples that have been lost, across all instances of data written for the <em>Topic</em>.</td>
</tr>
<tr>
<td></td>
<td>total_count_change</td>
<td>The incremental number of DDS samples lost since the last time the <em>Listener</em> was called or the status was read.</td>
</tr>
<tr>
<td>DDS_SampleLostStatusKind</td>
<td>last_reason</td>
<td>The reason the last DDS sample was lost. See Table 8.11 DDS_SampleLostStatusKind.</td>
</tr>
</tbody>
</table>

The reason the DDS sample was lost appears in the *last_reason* field. The possible values are listed in Table 8.11 DDS_SampleLostStatusKind.

**Table 8.11 DDS_SampleLostStatusKind**

<table>
<thead>
<tr>
<th>Reason Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT_LOST</td>
<td>The sample was not lost.</td>
</tr>
<tr>
<td>LOST_BY_AVAILABILITY_WAITING_TIME</td>
<td>max_data_availability_waiting_time in the 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370 expired.</td>
</tr>
<tr>
<td>LOST_BY_DECODING_FAILURE</td>
<td>When using BEST EFFORT in the 7.5.21 RELIABILITY QosPolicy on page 447, a sample was lost because it could not be decoded.</td>
</tr>
<tr>
<td></td>
<td>When using RELIABLE in the RELIABILITY QosPolicy, the sample is rejected, not lost, with the reason REJECTED_BY_DECODING_FAILURE.</td>
</tr>
<tr>
<td>LOST_BY_DESERIALIZATION_FAILURE</td>
<td>A sample was lost because it could not be deserialized. A sample may fail to be deserialized for the following reasons:</td>
</tr>
<tr>
<td></td>
<td>- The subscribing application has received a sample with a sequence or string member that is longer than the maximum allowed by the DataReader’s data type.</td>
</tr>
<tr>
<td></td>
<td>- The subscribing application has received a sample with an unknown enum value. See the description of the dds.sample_assignability.accept_unknown_enum_value property in the Property Reference Guide for more information.</td>
</tr>
<tr>
<td></td>
<td>- The subscribing application has received a sample with an unknown union discriminator value. See the description of the dds.sample_assignability.accept_unknown_union_discriminator property in the Property Reference Guide for more information.</td>
</tr>
<tr>
<td></td>
<td>- The subscribing application has received a sample with an out-of-range value for one of the members that has been configured with a minimum or maximum value using the min, max, or range type annotations.</td>
</tr>
<tr>
<td></td>
<td>- Sample corruption has occurred. If this is the case, then using RTI Security Plugins enabling CRC (see the compute_crc and check_crc fields in the 9.5.9 WIRE PROTOCOL QosPolicy (DDS Extension) on page 670) can help avoid these failures.</td>
</tr>
</tbody>
</table>
Table 8.11 DDS_SampleLostStatusKind

<table>
<thead>
<tr>
<th>Reason Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOST_BY_INCOMPLETE_COHERENT_SET</td>
<td>A sample was lost because it is part of an incomplete coherent set. An incomplete coherent set is a coherent set for which some of the samples are missing.</td>
</tr>
<tr>
<td></td>
<td>For example, consider a DataWriter using KEEP_LAST in the 7.5.12 HISTORY QosPolicy on page 420 with a depth of 1. The DataWriter publishes two samples of the same instance as part of a coherent set &quot;CS1&quot;; the first sample of &quot;CS1&quot; is replaced by a new sample before it can be successfully delivered to the DataReader. In this case, the coherent set containing the two samples is considered incomplete. The new sample, by default, will not be provided to the application, and will be reported as LOST_BY_INCOMPLETE_COHERENT_SET. (You can change this default behavior by setting drop_incomplete_coherent_set to FALSE in the 7.4.6 PRESENTATION QosPolicy on page 362. If you do, the new sample will be provided to the application, but it will be marked as part of an incomplete coherent set in the 8.4.6 The SampleInfo Structure on page 565.)</td>
</tr>
<tr>
<td>LOST_BY_INSTANCES_LIMIT</td>
<td>max_instances in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 was reached.</td>
</tr>
<tr>
<td>LOST_BY_LARGE_COHERENT_SET</td>
<td>A sample was lost because it was part of a large coherent set. A large coherent set is a coherent set that cannot fit all at once into the DataReader queue because resource limits are exceeded.</td>
</tr>
<tr>
<td></td>
<td>For example, if max_samples_per_instance on the DataReader is 10 and the coherent set has 15 samples for a given instance, the coherent set is a large coherent set that will be considered incomplete.</td>
</tr>
<tr>
<td></td>
<td>The resource limits that can lead to large coherent sets are: max_samples, max_samples_per_Instance, max_instances, and max_samples_per_remote_writer.</td>
</tr>
<tr>
<td>LOST_BY_OUT_OF_MEMORY</td>
<td>A sample was lost because there was not enough memory to store the sample.</td>
</tr>
<tr>
<td>LOST_BY_REMOTE_WRITERS_SAMPLES_</td>
<td>A resource limit on the number of samples published by a DataWriter on behalf of a virtual DataWriter that a DataReader may store was reached. (This field is currently not used.)</td>
</tr>
<tr>
<td>PER_VIRTUAL_QUEUE_LIMIT</td>
<td>max_remote_writers_per_instance in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576 was reached. (This limit is the number of Data Writers for a single instance from which a DataReader may read.)</td>
</tr>
<tr>
<td>LOST_BY_REMOTE_WRITERS_PER_INSTANCE</td>
<td>max_remote_writers_per_instance in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 was reached. (This limit is the number of Data Writers that are allowed to write the same sample.)</td>
</tr>
<tr>
<td>SAMPLE_LIMIT</td>
<td>When using BEST_EFFORT in the 7.5.21 RELIABILITY QosPolicy on page 447, max_samples in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 was reached.</td>
</tr>
<tr>
<td></td>
<td>When using RELIABLE in the RELIABILITY QosPolicy, reaching max_samples triggers a rejection, not a loss, with the reason REJECTED_BY_SAMPLES_LIMIT.</td>
</tr>
<tr>
<td>LOST_BY_SAMPLES_LIMIT</td>
<td>When using BEST_EFFORT in the 7.5.21 RELIABILITY QosPolicy on page 447, max_samples_per_instance in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 was reached.</td>
</tr>
<tr>
<td></td>
<td>When using RELIABLE in the RELIABILITY QosPolicy, reaching max_samples_per_instance triggers a rejection, not a loss, with the reason REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT.</td>
</tr>
</tbody>
</table>
### Table 8.11 DDS_SampleLostStatusKind

<table>
<thead>
<tr>
<th>Reason Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOST_BY_SAMPLES_PER_REMOTE_WRITER_LIMIT</td>
<td>When using BEST_EFFORT in the 7.5.21 RELIABILITY QosPolicy on page 447, <strong>max_samples_per_remote_writer</strong> in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576 was reached. (This limit is the number of samples from a given DataWriter that a DataReader may store.) When using RELIABLE in the RELIABILITY QosPolicy, reaching <strong>max_samples_per_remote_writer</strong> triggers a rejection, not a loss, with the reason REJECTED_BY_SAMPLES_PER_REMOTE_WRITER_LIMIT.</td>
</tr>
<tr>
<td>LOST_BY_UNKNOWN_INSTANCE</td>
<td>A sample was lost because it didn’t contain enough information for the DataReader to know what instance it was associated with.</td>
</tr>
<tr>
<td>LOST_BY_VIRTUAL_WRITERS_LIMIT</td>
<td><strong>max_remote_virtual_writers</strong> in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576 was reached. (This limit is the number of virtual DataWriters from which a DataReader may read.)</td>
</tr>
</tbody>
</table>
### 8.3.7 Statuses for DataReaders

#### Table 8.11 DDS_SampleLostStatusKind

<table>
<thead>
<tr>
<th>Reason Kind</th>
<th>Description</th>
</tr>
</thead>
</table>
| LOST_BY_WRITER       | *A DataWriter removed the DDS sample before being received by the DataReader.*  
  
  The DataReader detects that a sample is lost:  
  
  - For Best Effort [7.5.21 RELIABILITY QosPolicy on page 447](#): once a sample with a higher sequence number is received.  
  - For Reliable RELIABILITY QosPolicy: once a heartbeat message is received that announces that a sample that the DataReader was waiting for is no longer available in the DataWriter’s queue (i.e., the first sequence number in the heartbeat is higher than the missing sample’s sequence number). Samples that are gapped through GAP messages are not considered lost.  
  
  Samples may be lost for any of the following reasons:  
  
  - The lifespan of a sample expired before it was received by a DataReader; see [7.5.14 LIFESPAN Qos Policy on page 425](#).  
  - For Best Effort RELIABILITY QosPolicy: a sample was lost on the network or arrived out of order at the DataReader. (For example, the DataReader received sample 2 but not sample 1; the DataReader considers sample 1 LOST_BY_WRITER.)  
  - For Reliable RELIABILITY QosPolicy:  
    - When using KEEP_LAST [7.5.12 HISTORY QosPolicy on page 420](#), unacknowledged samples can be overwritten if the history depth limit is reached for an instance.  
      重要：Depending on timing, samples that were replaced due to KEEP_LAST replacement may be gapped by a GAP message and are therefore not reported as lost by the DataReader, or, at other times, the heartbeat message will announce that the sample is no longer available, as described above, and these will be reported as lost.  
    - For KEEP_ALL HISTORY QosPolicy, the DataWriter can overwrite a sample in its queue after the DataReader was marked as ‘inactive’. Once a DataReader is marked as ‘inactive’, samples will no longer be considered unacknowledged by that DataReader until it becomes active again. This means that if resource limits are hit and space is needed for a new sample, an old sample may be replaced to make room even if the inactive DataReader never received it. A DataReader is considered inactive either because it is not making progress (see `inactivate_nonprogressing_readers`) or `max_heartbeat_retries` was exceeded.  

See [11.3.2 Tuning Queue Sizes and Other Resource Limits on page 696](#) for more information on changing sample loss or queue configuration.

The DataReaderListener’s `on_sample_lost()` callback is invoked when this status changes. You can also retrieve the value by calling the DataReader’s `get_sample_lost_status()` operation; this will also reset the status so it is no longer considered “changed.”

#### 8.3.7.8 SAMPLE_REJECTED Status

This status indicates that one or more DDS samples received from a matched DataWriter have been rejected by the DataReader because a resource limit would have been exceeded: for example, if the receive...
8.3.7 Statuses for DataReaders

queue is full because the number of DDS samples in the queue is equal to the max_samples parameter of the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451. These rejected samples could be accepted later once the conditions for acceptance are met (e.g., once the number of samples in the queue becomes less than max_samples). A sample that is rejected can be resent any number of times until it is eventually reported as lost, dropped, or accepted.

Samples can be rejected only with reliable communication; see 7.5.21 RELIABILITY QosPolicy on page 447. In best-effort communication, samples cannot be rejected because samples cannot be received again and are not eligible for resending.

The structure for the rejected status appears in Table 8.12 DDS_SampleRejectedStatus. The reason the DDS sample was rejected appears in the last_reason field. The possible values are listed in Table 8.13 DDS_SampleRejectedStatusKind.

Table 8.12 DDS_SampleRejectedStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative count of all the DDS samples that have been rejected by the DataReader.</td>
</tr>
<tr>
<td></td>
<td>total_count_change</td>
<td>The incremental number of DDS samples rejected since the last time the Listener was called or the status was read.</td>
</tr>
<tr>
<td></td>
<td>current_count</td>
<td>The current number of writers with which the DataReader is matched.</td>
</tr>
<tr>
<td></td>
<td>current_count_change</td>
<td>The change in current_count since the last time the Listener was called or the status was read.</td>
</tr>
<tr>
<td>DDS_SampleRejectedStatusKind</td>
<td>last_reason</td>
<td>Reason for rejecting the last DDS sample. See Table 8.13 DDS_SampleRejectedStatusKind.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>last_instance_handle</td>
<td>Handle to the data-instance for which the last DDS sample was rejected.</td>
</tr>
</tbody>
</table>

Table 8.13 DDS_SampleRejectedStatusKind

<table>
<thead>
<tr>
<th>Reason Kind</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_NOT_REJECTED</td>
<td>DDS sample was accepted.</td>
</tr>
<tr>
<td>REJECTED_BY_DECODER_FAILURE</td>
<td>When using RELIABLE in the 7.5.21 RELIABILITY QosPolicy on page 447, a sample was rejected because it could not be decoded.</td>
</tr>
<tr>
<td></td>
<td>When using BEST_EFFORT in the 7.5.21 RELIABILITY QosPolicy on page 447, the sample is lost, not rejected, with the reason LOST_BY_DECODER_FAILURE.</td>
</tr>
<tr>
<td>DDS_REJECTED_BY_INSTANCES_LIMIT</td>
<td>This field is not currently used.</td>
</tr>
</tbody>
</table>
The `DataReaderListener`'s `on_sample_rejected()` callback is invoked when this status changes. You can also retrieve the value by calling the `DataReader`'s `get_sample_rejected_status()` operation; this will also reset the status so it is no longer considered “changed.”

### 8.3.7.9 SUBSCRIPTION_MACHED Status

A change to this status indicates that the `DataReader` discovered a matching `DataWriter`. A ‘match’ occurs only if the `DataReader` and `DataWriter` have the same `Topic`, same or compatible data type, and compatible QoS policies. (For more information on compatible data types, see the [RTI Connext DDS Core Libraries Extensible Types Guide](http://doc.rti.org/1.5/dds/Extensible.html) ) In addition, if user code has directed Connext DDS to ignore certain `DataWriters`, then those `DataWriters` will never be matched. See 18.4.2 Ignoring Publications and Subscriptions on page 843 for more on setting up a `DomainParticipant` to ignore specific `DataWriters`.

This status is also changed (and the listener, if any, called) when a match is ended. A `DataReader` will become unmatched from a `DataWriter` when that `DataWriter` goes away for any of the following reasons:

- The `DomainParticipant` containing the matched `DataWriter` has lost liveliness.
- The `DataReader` or the matched `DataWriter` has changed QoS such that the entities are now incompatible.
- The matched `DataWriter` has been deleted.
This status may reflect changes from multiple match or unmatch events, and the `current_count_change` can be used to determine the number of changes since the listener was called back or the status was checked.

The structure for this status appears in Table 8.14 DDS_SubscriptionMatchedStatus.

**Table 8.14 DDS_SubscriptionMatchedStatus**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>total_count</td>
<td>Cumulative number of times the DataReader discovered a &quot;match&quot; with a DataWriter. This number increases whenever a new match is discovered. It does not decrease when an existing match goes away for any of the reasons listed above.</td>
</tr>
<tr>
<td></td>
<td>total_count_change</td>
<td>The changes in <code>total_count</code> since the last time the listener was called or the status was read. Note that this number will never be negative (because it's the total number of times the DataReader ever matched with a DataWriter).</td>
</tr>
<tr>
<td></td>
<td>current_count</td>
<td>The number of DataWriters currently matched to the concerned DataReader. This number increases when a new match is discovered and decreases when an existing match goes away for any of the reasons listed above.</td>
</tr>
<tr>
<td></td>
<td>current_count_change</td>
<td>The change in <code>current_count</code> since the last time the listener was called or the status was read. Note that a negative <code>current_count_change</code> means that one or more DataWriters have become unmatched for one or more of the reasons listed above.</td>
</tr>
<tr>
<td></td>
<td>current_count_peak</td>
<td>Greatest number of DataWriters that matched this DataReader simultaneously. That is, there was no moment in time when more than this many DataWriters matched this DataReader. (As a result, <code>total_count</code> can be higher than <code>current_count_peak</code>.)</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>last_publication_handle</td>
<td>This InstanceHandle can be used to look up which remote DataWriter was the last to cause this DataReader's status to change, using the DataReader's <code>get_matched_publication_data()</code> method. If the DataWriter no longer matches this DataReader due to any of the reasons listed above except incompatible QoS, then the DataWriter has been purged from this DataReader's DomainParticipant discovery database. (See 15.1 What is Discovery? on page 765.) In that case, the DataReader's <code>get_matched_publication_data</code> method will not be able to return information about the DataWriter. The only way to get information about the lost DataWriter is if you cached the information previously.</td>
</tr>
</tbody>
</table>

The DataReaderListener's `on_subscription_matched()` callback is invoked when this status changes. You can also retrieve the value by calling the DataReader’s `get_subscription_match_status()` operation; this will also reset the status so it is no longer considered “changed.”

### 8.3.8 Accessing and Managing Instances (Working with Keyed Data Types)

This section describes how instances work on DataReaders. This section applies only to data types that use keys; see 2.4 DDS Samples, Instances, and Keys on page 18. See also Chapter 6 Working with Instances on page 257.
A DataReader receives updates about instances and instance state changes as DATAAVAILABLE statuses, the same way it receives data updates. (See 8.3.7.1 DATAAVAILABLE Status on page 517.) DataReaders can access instance state as part of the SampleInfo that is returned when calling any variant of read() or take() (such as read_instance() or take_instance()).

### 8.3.8.1 Instance States

As seen in Figure 8.11: Instance States on the next page, Connext DDS keeps an instance_state for each instance:

- **ALIVE**: The following are all true: (a) DDS samples have been received for the instance, (b) there are live DataWriters writing the instance, and (c) the instance has not been explicitly disposed (or more DDS samples have been received after it was disposed).

- **NOT_ALIVE_DISPOSED**: The instance was explicitly disposed by a DataWriter by means of the dispose() operation, or implicitly as a result of the autodispose_unregistered_instances QoS setting.

- **NOT_ALIVE_NO_WRITERS**: The instance has been declared as not-alive by the DataReader because it has determined that there are no live DataWriter entities that have previously written the instance.

Instances can cycle through these phases as seen in the state diagram below, becoming NOT_ALIVE and then becoming ALIVE again. To track these transitions, there is metadata the DataReader can query called generation counts. (See 8.3.8.2 Generation Counts and Ranks on page 537.)

The events that cause the instance_state to change can depend on the setting of the 7.5.17 OWNERSHIP QosPolicy on page 434:

- If OWNERSHIP QoS is set to EXCLUSIVE, the instance_state becomes NOT_ALIVE_DISPOSED only if the DataWriter that currently “owns” the instance explicitly disposes it. The instance_state will become ALIVE again only if the DataWriter that owns the instance writes it. Note that ownership of the instance is determined by a combination of the OWNERSHIP QoSPolicy and 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438. Ownership of an instance can dynamically change.

- If OWNERSHIP QoS is set to SHARED, the instance_state becomes NOT_ALIVE_DISPOSED if any DataWriter explicitly disposes the instance. The instance_state becomes ALIVE as soon as any DataWriter writes the instance again.
Since the `instance_state` in the `SampleInfo` structure is a per-instance concept, all DDS data samples related to the same instance that are returned by `read()` or `take()` will have the same value for `instance_state`. This means that if there are samples for that instance in the `DataReader`’s queue that were received when the instance was `ALIVE`, and a subsequent dispose message is received, the samples’ metadata will indicate that the instance’s state is `NOT_ALIVE_DISPOSED` in all of them.

**Note:** The `instance_state` always reflects the current state of the instance at the time of reading.
8.3.8 Accessing and Managing Instances (Working with Keyed Data Types)

Figure 8.12: Before and After Dispose Received

<table>
<thead>
<tr>
<th>SampleInfo</th>
<th>Data</th>
<th>Before “disposed” received</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIVE</td>
<td>ALIVE</td>
<td></td>
</tr>
<tr>
<td>valid_data = true</td>
<td>valid_data = true</td>
<td></td>
</tr>
<tr>
<td>flightId = 265</td>
<td>flightId = 265</td>
<td></td>
</tr>
<tr>
<td>currentPosition =</td>
<td>currentPosition =</td>
<td></td>
</tr>
<tr>
<td>{ [37, 20], [121, 53] }</td>
<td>{ [37, 21], [121, 52] }</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SampleInfo</th>
<th>Data</th>
<th>After “disposed” received</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT_ALIVE_DISPOSED</td>
<td>NOT_ALIVE_DISPOSED</td>
<td></td>
</tr>
<tr>
<td>valid_data = true</td>
<td>valid_data = true</td>
<td></td>
</tr>
<tr>
<td>flightId = 265</td>
<td>flightId = 265</td>
<td>X</td>
</tr>
<tr>
<td>currentPosition =</td>
<td>currentPosition =</td>
<td></td>
</tr>
<tr>
<td>{ [37, 20], [121, 53] }</td>
<td>{ [37, 21], [121, 52] }</td>
<td></td>
</tr>
</tbody>
</table>

*When the dispose message is received (the box with the X, with valid_data = false), all samples for the flight 265 instance in the queue are marked as NOT_ALIVE_DISPOSED, even those that contain live data from when the instance was ALIVE.*

**Note:** If an instance transitions its state to NOT_ALIVE_NO_WRITERS due to one or more DataWriters losing liveliness, it will not transition back to ALIVE if the DataWriter regains liveliness. It only returns to the ALIVE state if a DataWriter writes a new sample of the instance.

### 8.3.8.2 Generation Counts and Ranks

Generation counts and ranks allow your application to distinguish DDS samples belonging to different ‘generations’ of the instance. It is possible for an instance to become alive, be disposed and become not-alive, and then cycle again from alive to not-alive states during the operation of an application. Each time an instance becomes alive defines a new generation for the instance.

It is possible that an instance may cycle through alive and not-alive states multiple times before the application accesses the DDS data samples for the instance. This means that the DDS data samples returned by `read()` and `take()` may cross generations. That is, some DDS samples were published when the instance was alive in one generation and other DDS samples were published when the instance transitioned through the non-alive state into the alive state again. It may be important to your application to distinguish the DDS data samples by the generation in which they were published.
Each `DataReader` keeps two counters for each instance it detects (recall that instances are distinguished by their key values):

- **disposed_generation_count**: Counts how many times the `instance_state` of the corresponding instance changes from `NOT_ALIVE_DISPOSED` to `ALIVE`.
- **no_writers_generation_count**: Counts how many times the `instance_state` of the corresponding instance changes from `NOT_ALIVE_NO_WRITERS` to `ALIVE`.

The `disposed_generation_count` and `no_writers_generation_count` fields in the `SampleInfo` structure capture a snapshot of the corresponding counters at the time the corresponding DDS sample was received.

The `sample_rank` and `generation_rank` in the `SampleInfo` structure are computed relative to the sequence of DDS samples returned by `read()` or `take()`:

- **sample_rank**: Indicates how many DDS samples of the same instance follow the current one in the sequence. The DDS samples are always time-ordered, thus the newest DDS sample of an instance will have a `sample_rank` of 0. Depending on what you have configured `read()` and `take()` to return (by passing in state masks and through the `max_samples_per_read` field in 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576), a `sample_rank` of 0 may or may not be the newest DDS sample that was ever received. It is just the newest DDS sample in the sequence that was returned. The `sample_rank` value could be used by an application to determine that there are newer samples in the sequence and that it might want to skip processing the older samples.

- **generation_rank**: Indicates the difference in ‘generations’ between the DDS sample and the newest DDS sample of the same instance as returned in the sequence. If a DDS sample belongs to the same generation as the newest DDS sample in the sequence returned by `read()` and `take()`, then `generation_rank` will be 0.

- **absolute_generation_rank**: Indicates the difference in ‘generations’ between the DDS sample and the newest DDS sample of the same instance ever received by the `DataReader`. Recall that the data sequence returned by `read()` and `take()` may not contain all of the data in the `DataReader`’s receive queue. Thus, a DDS sample that belongs to the newest generation of the instance will have an `absolute_generation_rank` of 0.

By using the `sample_rank`, `generation_rank` and `absolute_generation_rank` information in the `SampleInfo` structure, your application can determine exactly what happened to the instance and thus make appropriate decisions of what to do with the DDS data samples received for the instance. For example:

- A DDS sample with `sample_rank`= 0 is the newest DDS sample of the instance in the returned sequence.
8.3.8 Accessing and Managing Instances (Working with Keyed Data Types)

- DDS samples that belong to the same generation will have the same `generation_rank` (as well as `absolute_generation_rank`).
- DDS samples with `absolute_generation_rank = 0` belong to the newest generation for the instance received by the `DataReader`.

The ‘generation count’ and ‘rank’ values are statistics that are locally generated by each `DataReader` and maintained as part of the metadata for the instance that they refer to. Therefore, if the instance is reclaimed and then returns at a later point in time, these counters will all restart at 0.

### 8.3.8.3 Valid Data Flag

The `SampleInfo` structure’s `valid_data` flag indicates whether the DDS sample contains data or is only used to communicate a change in the `instance_state` of the instance.

Normally, each DDS sample contains both a `SampleInfo` structure and some data. However, there are situations in which the DDS sample only contains the `SampleInfo` and does not have any associated data. This occurs when `Connext DDS` notifies the application of a change of state for an instance for which there is no associated data. An example is when `Connext DDS` detects that an instance has no writers and changes the corresponding `instance_state` to `NOT_ALIVE_NO_WRITERS`.

If the `valid_data` flag is `TRUE`, then the DDS sample contains valid data. If the flag is `FALSE`, the DDS sample contains no data.

To ensure correctness and portability, your application must check the `valid_data` flag prior to accessing the data associated with the DDS sample, and only access the data if it is `TRUE`. The value of data is undefined when the `valid_data` flag is false.

### 8.3.8.4 Looking up an Instance Handle

Some operations, such as `read_instance()`, require an `instance_handle` parameter. If you need to get such a handle, you can call the `FooDataReader`’s `lookup_instance()` operation, which takes a sample with key fields specified as a parameter and returns a handle to that instance.

```c
DDS_InstanceHandle_t lookup_instance (const Foo & key_holder)
```

The instance must have been received by the `DataReader` in order for the `DataReader` to look it up. If the instance is not known to the `DataReader`, this operation returns `DDS_HANDLE_NIL`.

### 8.3.8.5 Getting the Key Value for an Instance

Once you have an instance handle (using `lookup_instance()`, as part of a status change notification, or through the `SampleInfo`), you can use the `DataReader`’s `get_key_value()` operation to retrieve the value of the key of the corresponding instance. The key fields of the data structure passed into `get_key_value()` will be filled out with the original values used to generate the instance handle. The key fields are defined...
when the data type is defined; see 2.4 DDS Samples, Instances, and Keys on page 18 for more information.

If you set `propagate_dispose_of_unregistered_instances` to true and wish to call `get_key_value()` for instances for which only a dispose sample has been received, the `serialize_key_with_dispose` field in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 must be set to true.

### 8.3.8.6 Instance Resource Limits and Memory Management

In Connext DDS, memory is primarily pre-allocated when creating entities. When data is keyed, the memory associated with each instance used for storing instance-specific metadata is allocated when the `DataReader` is created. Memory is not freed at runtime, unless you delete an entity. Instead, memory is made available to be reused by the `DataReader`, or “reclaimed”.

The `DataReader` can receive a number of instances defined by the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 and 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576. It is also important to understand that an instance in the `DataReader` queue has two parts that make up the instance metadata: an active state and a minimum state. The resource limits control the amount of active state and minimum state that should be maintained. (Note: the concept of active and minimum state does not apply to instance metadata in the `DataWriter` queue.)

#### 8.3.8.6.1 Active State and Minimum State

An instance is considered either *attached* or *detached* in the `DataReader` queue and is composed of two parts, which make up the instance metadata: an active state and a minimum state.
Figure 8.13: Active and Minimum Instance States

An instance is considered *attached* when the *DataReader* is actively managing all possible state that can be associated with an instance, including the associated samples, the instance and view states, generation and sample ranks, the list of remote writers that are known to be writing the instance, and so on. Only attached instances can have associated samples. A *DataReader* keeps both the active and the minimum state for attached instances. The sum of the `alive_instance_count`, `disposed_instance_count`, and `no_writers_instance_count` statistics in the 8.3.7.2 DATA_READER_CACHE_STATUS on page 517 reflects the total number of attached instances currently in the *DataReader* queue.

The following is applicable only if `keep_minimum_state_forInstances` in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576 is TRUE (by default, it is). See 8.6.2.2 keep_minimum_state_forInstances on page 582 for more on this QoS setting.

An instance is considered *detached* when the *DataReader* is only maintaining the minimum state for the instance. When instances are replaced or purged from the *DataReader* queue, by default only the active state of the instance is reclaimed. A minimum amount of state for the instance is kept even after the instance is removed in order to maintain system consistency without having to waste resources (memory and CPU) by keeping other information around that is no longer relevant (i.e., the active state). The minimum state is used when instances that have been removed re-enter the system. This can happen, for example, when a non-VOLATILE *DataReader* and *DataWriter* lose liveliness and then re-discover each other. The *DataWriter* will resend its history, but if the *DataReader* has the minimum state information for any instances that it removed during the disconnection, the previously received duplicate samples will be...
filtered out and dropped before being accepted into the DataReader’s queue again. The minimum state includes information such as the last source timestamp, the keyhash, and the list of virtual writers for the instance. In general, you should keep keep_minimum_state_for_instances set to true if you are using the Durable Reader State, MultiChannel DataWriters, or RTI Persistence Service, or in any system where instances may be removed and then re-enter the system either because the original DataWriter is re-discovered or writes the instance again or a new DataWriter begins writing the instance.

An instance transitions from what is considered an attached instance to a detached instance when the instance is removed from the DataReader queue (purged or replaced). This can happen under the following conditions:

- The instance is replaced due to the instance_replacement settings in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576.
- There are no more samples associated with the instance. Samples can be removed from the DataReader queue through the use of the take() operation, or various QoS configurations such as a finite lifespan or KEEP_LAST history configuration. In addition, at least one of the following must be true:
  - The instance was in the NOT_ALIVE_NO_WRITERS instance state and autopurge_nowriter_instances_delay has expired. (The default value for the autopurge_nowriter_instances_delay is 0, so by default instances are purged as soon as the instance is empty and transitions to NOT_ALIVE_NO_WRITERS.)
  - The instance was in the NOT_ALIVE_DISPOSED instance state and the autopurge_disposed_instances_delay has expired.

The detached_instance_count statistic in the 8.3.7.2 DATA_READER_CACHE_STATUS on page 517 counts the total number of detached instances currently in the DataReader queue.

### 8.3.8.6.2 Instance Resource Limit QoS Policies

The 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 and 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576 include the following fields that affect the number of instances that can be received:

- **max_instances** (7.5.22 RESOURCE_LIMITS QosPolicy on page 451): A resource limit on the number of attached instances that can be managed by Connext DDS. By default, max_instances is UNLIMITED, so you are bounded only by the physical resources of your system. If the max_instances limit has been hit, and a sample is received for a new instance, Connext DDS will first attempt to replace an instance according to what you have configured in the instance_replacement field in the 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576. If there are not any replaceable instances (by default empty NOT_ALIVE_DISPOSED and NOT_ALIVE_NO_WRITERS instances are replaceable, and ALIVE instances are not
When keep minimum state for instances is not exceeded, Connext DDS will check max instances. If max instances is exceeded, Connext DDS will replace one of the detached instances with the new, attached one. The application could receive duplicate samples for the replaced instance if it becomes alive again.

max total instances should be equal to the number of attached instances you want to keep, plus the number of detached instances you want to keep.

keep minimum state for instances (8.6.2 DATA_READER RESOURCE LIMITS QosPolicy (DDS Extension) on page 576): This QoS setting can be used to enable or disable Connext DDS from keeping minimum instance information for detached instances. By default, this setting is TRUE. This minimum instance information is useful for the features described earlier in this section. If this QoS setting is FALSE, minimum instance state will not be kept, and therefore detached instances will not be kept.

The instance replacement field in the 8.6.2 DATA_READER RESOURCE LIMITS QosPolicy (DDS Extension) on page 576 controls whether instances can be replaced to make room for new ones. See 8.6.2.3 Configuring DataReader Instance Replacement on page 583.

The 8.6.3 READER_DATA LIFECYCLE QoS Policy on page 586 controls whether the DataReader can remove data from the queue if instance state becomes NOT_ALIVE_NO_WRITERS or NOT_ALIVE_DISPOSED.

### 8.3.9 Setting DataReader QosPolicies

A DataReader’s QosPolicies control its behavior. Think of QosPolicies as the 'properties' of a DataReader.

The DDS DataReaderQos structure has the following format:

```c
DDS_DataReaderQos struct {
    DDS_DurabilityQosPolicy durability;
};
```
8.3.9 Setting DataReader QosPolicies

```c
DDS_DeadlineQosPolicy  deadline;
DDS_LatencyBudgetQosPolicy  latency_budget;
DDS_LivelinessQosPolicy  liveliness;
DDS_ReliabilityQosPolicy  reliability;
DDS_DestinationOrderQosPolicy  destination_order;
DDS_HistoryQosPolicy  history;
DDS_ResourceLimitsQosPolicy  resource_limits;
DDS_UserDataQosPolicy  user_data;
DDS_OwnershipQosPolicy  ownership;
DDS_TimeBasedFilterQosPolicy  time_based_filter;
DDS_ReaderDataLifecycleQosPolicy  reader_data_lifecycle;
DDS_DataRepresentationQosPolicy  representation;
DDS_TypeConsistencyEnforcementQosPolicy  type_consistency;
DDS_DataTagQosPolicy  data_tags;
// extensions to the DDS standard:
DDS_ReaderResourceLimitsQosPolicy  reader_resource_limits;
DDS_DataReaderProtocolQosPolicy  protocol;
DDS_TransportSelectionQosPolicy  transport_selection;
DDS_TransportUnicastQosPolicy  unicast;
DDS_TransportMulticastQosPolicy  multicast;
DDS_PropertyQosPolicy  property;
DDS_ServiceQosPolicy  service;
DDS_AvailabilityQosPolicy  availability;
DDS_EntityNameQosPolicy  subscription_name;
DDS_TransportPriorityQosPolicy  transport_priority;
DDS_TypeSupportQosPolicy  type_support;
} DDS_DataReaderQos;
```

**Note:** `set_qos()` cannot always be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

Table 8.15 DataReader QosPolicies summarizes the meaning of each policy. (They appear alphabetically in the table.) For information on why you would want to change a particular QosPolicy, see the referenced section. For defaults and valid ranges, please refer to the API Reference HTML documentation.

### Table 8.15 DataReader QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Availability</strong></td>
<td>This QoS policy is used in the context of two features:</td>
</tr>
<tr>
<td></td>
<td>- For a Collaborative DataWriter, specifies the group of DataWriters expected to collaboratively provide data and the timeouts that control when to allow data to be available that may skip DDS samples.</td>
</tr>
<tr>
<td></td>
<td>- For a Durable Subscription, configures a set of Durable Subscriptions on a DataWriter.</td>
</tr>
<tr>
<td></td>
<td>See 7.5.1 <strong>AVAILABILITY QosPolicy (DDS Extension)</strong> on page 370.</td>
</tr>
<tr>
<td><strong>DataReaderProtocol</strong></td>
<td>This QosPolicy configures the DDS on-the-network protocol, RTPS. See 8.6.1 <strong>DATAADER_PROTOCOL QosPolicy (DDS Extension)</strong> on page 570.</td>
</tr>
</tbody>
</table>
### Table 8.15 DataReader QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataReaderResourceLimits</td>
<td>Various settings that configure how DataReaders allocate and use physical memory for internal resources. See 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576.</td>
</tr>
<tr>
<td>DataRepresentation</td>
<td>Specifies which versions of the Extended Common Data Representation (CDR) are requested. See 7.5.3 DATA_REPRESENTATION QosPolicy on page 380.</td>
</tr>
<tr>
<td>DataTag</td>
<td>A sequence of (name, value) string pairs that may be used by the Access Control plugin. See 7.5.4 DATATAG QosPolicy on page 388.</td>
</tr>
<tr>
<td>Deadline</td>
<td>For a DataReader, it specifies the maximum expected elapsed time between arriving DDS data samples. For a DataWriter, it specifies a commitment to publish DDS samples with no greater elapsed time between them. See 7.5.7 DEADLINE QosPolicy on page 405.</td>
</tr>
<tr>
<td>DestinationOrder</td>
<td>Controls how Connext DDS will deal with data sent by multiple DataWriters for the same topic. Can be set to &quot;by reception timestamp&quot; or to &quot;by source timestamp&quot;. See 7.5.8 DESTINATION_ORDER QosPolicy on page 408.</td>
</tr>
<tr>
<td>Durability</td>
<td>Specifies whether or not Connext DDS will store and deliver data that were previously published to new DataReaders. See 7.5.9 DURABILITY QosPolicy on page 411.</td>
</tr>
<tr>
<td>DurabilityService</td>
<td>Various settings to configure the external Persistence Service used by Connext DDS for DataWriters with a Durability QoS setting of Persistent Durability. See 7.5.10 DURABILITY SERVICE QosPolicy on page 416.</td>
</tr>
<tr>
<td>EntityName</td>
<td>Assigns a name to a DataReader. See 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
<tr>
<td>History</td>
<td>Specifies how much data must to be stored by Connext DDS for the DataWriter or DataReader. This QosPolicy affects the 7.5.21 RELIABILITY QosPolicy on page 447 as well as the 7.5.9 DURABILITY QosPolicy on page 411. See 7.5.12 HISTORY QosPolicy on page 420.</td>
</tr>
<tr>
<td>LatencyBudget</td>
<td>Suggestion to Connext DDS on how much time is allowed to deliver data. See 7.5.13 LATENCYBUDGET QoS Policy on page 425.</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Specifies and configures the mechanism that allows DataReaders to detect when DataWriters become disconnected or &quot;dead.&quot; See 7.5.15 LIVELINESS QosPolicy on page 427.</td>
</tr>
<tr>
<td>Property</td>
<td>Stores name/value (string) pairs that can be used to configure certain parameters of Connext DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.</td>
</tr>
<tr>
<td>ReaderDataLifecycle</td>
<td>Controls how a DataReader manages the lifecycle of the data that it has received. See 8.6.3 READER_DATA_LIFECYCLE QoS Policy on page 586.</td>
</tr>
<tr>
<td>Reliability</td>
<td>Specifies whether or not Connext DDS will deliver data reliably. See 7.5.21 RELIABILITY QosPolicy on page 447.</td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>Controls the amount of physical memory allocated for entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.</td>
</tr>
<tr>
<td>Service</td>
<td>Intended for use by RTI infrastructure services. User applications should not modify its value. See 7.5.23 SERVICE QosPolicy (DDS Extension) on page 454.</td>
</tr>
</tbody>
</table>
### Table 8.15 DataReader QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TimeBasedFilter</td>
<td>Set by a DataReader to limit the number of new data values received over a period of time. See 8.6.4 TIME_BASED_FILTER QosPolicy on page 588.</td>
</tr>
<tr>
<td>TransportMulticast</td>
<td>Specifies the multicast address on which a DataReader wants to receive its data. Can specify a port number as well as a subset of the available transports. See 8.8.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591.</td>
</tr>
<tr>
<td>TransportPriority</td>
<td>Set by a DataReader to tell Connext DDS that the data being sent is a different &quot;priority&quot; than other data. See 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458.</td>
</tr>
<tr>
<td>TransportSelection</td>
<td>Allows you to select which physical transports a DataWriter or DataReader may use to send or receive its data. See 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459.</td>
</tr>
<tr>
<td>TransportUnicast</td>
<td>Specifies a subset of transports and port number that can be used by an Entity to receive data. See 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460.</td>
</tr>
<tr>
<td>TypeConsistencyEnforcement</td>
<td>Defines rules that determine whether the type used to publish a given data stream is consistent with that used to subscribe to it. See 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594.</td>
</tr>
<tr>
<td>TypeSupport</td>
<td>Used to attach application-specific value(s) to a DataWriter or DataReader. These values are passed to the serialization or deserialization routine of the associated data type. Also controls whether padding bytes are set to 0 during serialization. See 7.5.29 TYPESUPPORT QosPolicy (DDS Extension) on page 464.</td>
</tr>
<tr>
<td>UserData</td>
<td>Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to Connext DDS's discovery meta-data. See 7.5.30 USER_DATA QosPolicy on page 465.</td>
</tr>
</tbody>
</table>

For a DataReader to communicate with a DataWriter, their corresponding QosPolicies must be compatible. For QosPolicies that apply both to the DataWriter and the DataReader, the setting in the DataWriter is considered what the DataWriter “offers” and the setting in the DataReader is what the DataReader “requests.” Compatibility means that what is offered by the DataWriter equals or surpasses what is requested by the DataReader. See 4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property on page 180.

Some of the policies may be changed after the DataReader has been created. This allows the application to modify the behavior of the DataReader while it is in use. To modify the QoS of an already-created DataReader, use the get_qos() and set_qos() operations on the DataReader. This is a general pattern for all Entities, described in 4.1.7.3 Changing the QoS for an Existing Entity on page 175.

### 8.3.9.1 Configuring QoS Settings when the DataReader is Created

As described in 8.3.1 Creating DataReaders on page 510, there are different ways to create a DataReader, depending on how you want to specify its QoS (with or without a QoS Profile).

- In Figure 8.9: Creating a DataReader with Default QosPolicies on page 511, there is an example of how to create a DataReader with default QosPolicies by using the special constant, DDS_DATAREADER_QOS_DEFAULT, which indicates that the default QoS values for a
8.3.9 Setting DataReader QoS Policies

DataReader should be used. The default DataReader QoS values are configured in the Subscriber or DomainParticipant; you can change them with set_default_datareader_qos() or set_default_datareader_qos_with_profile(). Then any DataReaders created with the Subscriber will use the new default values. As described in 4.1.7 Getting, Setting, and Comparing QoS Policies on page 172, this is a general pattern that applies to the construction of all Entities.

- To create a DataReader with non-default QoS without using a QoS Profile, see the example code in Figure 8.14: Creating a DataReader with Modified QoS Policies (not from a profile) below. It uses the Subscriber’s get_default_reader_qos() method to initialize a DDS_DataReaderQos structure. Then the policies are modified from their default values before the structure is used in the create_datareader() method.

- You can also create a DataReader and specify its QoS settings via a QoS Profile. To do so, you will call create_datareader_with_profile(), as seen in Figure 8.15: Creating a DataReader with a QoS Profile on the next page.

- If you want to use a QoS profile, but then make some changes to the QoS before creating the DataReader, call get_datawriter_qos_from_profile() and create_datawriter() as seen in Figure 8.16: Getting QoS Values from Profile, Changing QoS Values, Creating DataReader with Modified QoS Values on the next page.

For more information, see 7.3.1 Creating DataWriters on page 292 and Configuring QoS with XML (Chapter 19 on page 848).

Notes:

- The examples in this section use the Traditional C++ API; for examples in the Modern C++ API, see the sections "DataReader Use Cases," "Qos Use Cases," and "Qos Provider Use Cases" in the API Reference HTML documentation, under "Programming How-To's."

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

Figure 8.14: Creating a DataReader with Modified QoS Policies (not from a profile)

```cpp
DDS_DataReaderQos reader_qos;
// initialize reader_qos with default values
subscriber->get_default_datareader_qos(reader_qos);
// make QoS changes
reader_qos.history.depth = 5;
// Create the reader with modified qos
DDSDataReader * reader = subscriber->create_datareader(
    topic, reader_qos, NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL) {
    // ... error
}
// narrow it for your specific data type
FooDataReader* foo_reader = FooDataReader::narrow(reader);
```
8.3.9 Setting DataReader QosPolicies

Figure 8.15: Creating a DataReader with a QoS Profile

```
// Create the datareader
DDSDataReader * reader =
    subscriber->create_datareader_with_profile(
        topic, "MyReaderLibrary", "MyReaderProfile",
        NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL) {
    // ... error
};
// narrow it for your specific data type
FooDataReader* foo_reader = FooDataReader::narrow(reader);
```

Figure 8.16: Getting QoS Values from Profile, Changing QoS Values, Creating DataReader with Modified QoS Values

```
DDS_DataReaderQos reader_qos;
// Get reader QoS from profile
retcode = factory->get_datareader_qos_from_profile(
    reader_qos, "ReaderProfileLibrary", "ReaderProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes
reader_qos.history.depth = 5;
DDSDataReader * reader = subscriber->create_datareader(
    topic, reader_qos, NULL, DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    // handle error
}
```

8.3.9.2 Comparing QoS Values

The `equals()` operation compares two `DataReader`'s DDS_DataReaderQos structures for equality. It takes two parameters for the two `DataReader`'s QoS structures to be compared, then returns `TRUE` if they are equal (all values are the same) or `FALSE` if they are not equal.

8.3.9.3 Changing QoS Settings After the DataReader has been Created

There are two ways to change an existing `DataReader`'s QoS after it is has been created—again depending on whether or not you are using a QoS Profile.

- To change QoS programmatically (that is, without using a QoS Profile), use `get_qos()` and `set_qos()`. See the example code in Figure 8.17: Changing the QoS of Existing DataReader (without a QoS Profile) on the next page. It retrieves the current values by calling the `DataWriter`'s `get_qos()` operation. Then it modifies the value and calls `set_qos()` to apply the new value. Note, however, that some QosPolicies cannot be changed after the `DataWriter` has been enabled—this restriction is
noted in the descriptions of the individual QosPolicies.

- You can also change a DataReader’s (and all other Entities’) QoS by using a QoS Profile and calling set qos with profile(). For an example, see Figure 8.18: Changing the QoS of Existing DataReader with a QoS Profile below. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

Figure 8.17: Changing the QoS of Existing DataReader (without a QoS Profile)

```c
DDS_DataReaderQos reader_qos;
// Get current QoS.
if (datareader->get_qos(reader_qos) != DDS_RETCODE_OK) {
    // handle error
} // Makes QoS changes here
reader_qos.history.depth = 5;
// Set the new QoS
if (datareader->set_qos(reader_qos) != DDS_RETCODE_OK) {
    // handle error
}
```

Figure 8.18: Changing the QoS of Existing DataReader with a QoS Profile

```c
retcode = reader->set_qos_with_profile(
    "ReaderProfileLibrary","ReaderProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

8.3.9.4 Using a Topic’s QoS to Initialize a DataReader’s QoS

Several DataReader QosPolicies can also be found in the QosPolicies for Topics (see 5.1.3 Setting Topic QosPolicies on page 219). The QosPolicies set in the Topic do not directly affect the DataReaders (or DataWriters) that use that Topic. In many ways, some QosPolicies are a Topic-level concept, even though the DDS standard allows you to set different values for those policies for different DataWriters and DataReaders of the same Topic. Thus, the policies in the DDS_TopicQos structure exist as a way to help centralize and annotate the intended or suggested values of those QosPolicies. Connext DDS does not check to see if the actual policies set for a DataReader is aligned with those set in the Topic to which it is bound.

There are many ways to use the QosPolicies’ values set in the Topic when setting the QosPolicies’ values in a DataReader. The most straightforward way is to get the values of policies directly from the Topic and
use them in the policies for the `DataReader`, as shown in Figure 8.19: Copying Selected QoS from a Topic when Creating a DataReader below.

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QosPolicy Handling Considerations for C on page 182.

**Figure 8.19: Copying Selected QoS from a Topic when Creating a DataReader**

```c
DDS_DataReaderQos reader_qos;
DDS.TopicQos topic_qos;
// topic and publisher already created
// get current QoS for the topic, default QoS for the reader
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
if (publisher->get_default_datareader_qos(reader_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Copy specific policies from topic QoS to reader QoS
reader_qos.deadline = topic_qos.deadline;
reader_qos.reliability = topic_qos.reliability;
// Create the DataReader with the modified QoS
DDSDataReader* reader = publisher->create_datareader(topic,
    reader_qos,NULL, DDS_STATUS_MASK_NONE);
```

You can use the `Subscriber's copy_from_topic_qos()` operation to copy all of the common policies from the *Topic* QoS to a *DataReader* QoS. This is illustrated in Figure 8.20: Copying all QoS from a Topic when Creating a DataReader below.

**Figure 8.20: Copying all QoS from a Topic when Creating a DataReader**

```c
DDS_DataReaderQos reader_qos;
DDS.TopicQos topic_qos;
// topic, publisher, reader_listener already created
if (topic->get_qos(topic_qos) != DDS_RETCODE_OK) {
    // handle error
}
if (publisher->get_default_datareader_qos(reader_qos) != DDS_RETCODE_OK) {
    // handle error
}
// copy relevant QoS from topic into reader’s qos
publisher->copy_from_topic_qos(reader_qos, topic_qos);
// Optionally, modify policies as desired
reader_qos.deadline.duration.sec = 1;
reader_qos.deadline.duration.nanosec = 0;
// Create the DataReader with the modified QoS
```
The special macro, DDS_DATAREADER_QOS_USE_TOPIC_QOS, can be used to indicate that the DataReader should be created with the QoS that results from modifying the default DataReader QoS with the values specified by the Topic. See Figure 7.23: Combining Default Topic and DataWriter QoS (Option 1) on page 341 and Figure 7.24: Combining Default Topic and DataWriter QoS (Option 2) on page 342 for examples involving DataWriters. The same pattern applies to DataReaders.

For more information on the general use and manipulation of QosPolicies, see 4.1.7 Getting, Setting, and Comparing QosPolicies on page 172.

8.3.10 Navigating Relationships Among Entities

8.3.10.1 Finding Matching Publications

The following DataReader operations can be used to get information about the DataWriters that will send data to this DataReader. A publication consists of information about the DataWriter and its associated Publisher and Topic.

- `get_matched_publications()`
- `get_matched_publication_data()`

The `get_matched_publications()` operation will return a sequence of handles to matched DataWriters. You can use these handles in the `get_matched_publication_data()` method to get information about the DataWriter such as the values of its QosPolicies, as well as information about its Publisher and Topic.

Note that DataWriters that have been ignored using the DomainParticipant’s `ignore_publication()` operation are not considered to be matched even if the DataWriter has the same Topic and compatible QosPolicies. Thus, they will not be included in the list of DataWriters returned by `get_matched_publications()`. See 18.4.2 Ignoring Publications and Subscriptions on page 843 for more on `ignore_publication()`.

You can also get the DATA_READER_PROTOCOL_STATUS for matching publications with `get_matched_publication_datapoint_protocol_status()` (see 8.3.7.3 DATA_READER_PROTOCOL_STATUS on page 520).

Notes:

- The `get_matched_publications()` function includes the return of handles of matched DataWriters that are no longer alive. All of the handles returned by this function are valid inputs to the `get_matched_publication_data()` function.
- Status/data for a matched publication is kept even if the matched DataWriter is not alive. Status/data for a matched publication will be removed only if the DataWriter is gone: that is, the DataWriter is
destroyed and this change is propagated through a discovery update, or the DataWriter's DomainParticipant is gone (either gracefully or its liveness expired and Connext DDS is configured to purge not-alive participants). Once a matched DataWriter is gone, its status is deleted. If you try to get the status/data for a matched DataWriter that is gone, the 'get status' or 'get data' call will return an error.

- If you want to know which matched DataWriters are not alive, use is_matched_publication_alive (). See Table 8.3 DataReader Operations on page 505.

- The get_matched_publication_data() operation does not retrieve the type_code information from built-in-topic data structures. This information is available through the on_data_available() callback (if a DataReaderListener is installed on the PublicationBuiltinTopicDataDataReader).

See also: 8.3.10.2 Finding the Matching Publication’s ParticipantBuiltinTopicData below

### 8.3.10.2 Finding the Matching Publication's ParticipantBuiltinTopicData

get_matched_publication_participant_data() allows you to get the DDS_ParticipantBuiltinTopicData (see Table 18.1 Participant Built-in Topic’s Data Type (DDS_ParticipantBuiltinTopicData)) of a matched publication using a publication handle.

This operation retrieves the information on a discovered DomainParticipant associated with the publication that is currently matching with the DataReader.

The publication handle passed into this operation must correspond to a publication currently associated with the DataReader. Otherwise, the operation will fail with RETCODE_BAD_PARAMETERS. The operation may also fail with RETCODE_PRECONDITION_NOT_MET if the publication handle corresponds to the same DomainParticipant to which the DataReader belongs.

Use get_matched_publications() (see 8.3.10.1 Finding Matching Publications on the previous page) to find the publications that are currently matched with the DataReader.

### 8.3.10.3 Finding a DataReader's Related Entities

These DataReader operations are useful for obtaining a handle to various related entities:

- get_subscriber()

- get_topicdescription()

The get_subscriber() operation returns the Subscriber that created the DataReader. get_topicdescription() returns the Topic with which the DataReader is associated.

### 8.3.10.4 Looking Up an Instance Handle

Some operations, such as read_instance() and take_instance(), take an instance_handle parameter. If you need to get such as handle, you can call the lookup_instance() operation, which takes an instance as a
parameter and returns a handle to that instance.

8.3.10.5 Getting the Key Value for an Instance

If you have a handle to a data-instance, you can use the FooDataReader’s `get_key_value()` operation to retrieve the key for that instance. The value of the key is decomposed into its constituent fields and returned in aFoostructure. For information on keys and keyed data types, please see 2.4 DDS Samples, Instances, and Keys on page 18.

8.4 Using DataReaders to Access Data (Read & Take)

For user applications to access the data received for a `DataReader`, they must use the type-specific derived class or set of functions in the C API. Thus for a user data type ‘Foo’, you must use methods of the FooDataReader class. The type-specific class or functions are automatically generated if you use RTI Code Generator. Else, you will have to create them yourself, see 3.2.8 Type Codes for Built-in Types on page 74 for more details.

8.4.1 Using a Type-Specific DataReader (FooDataReader)

This section doesn't apply to the Modern C++ API, where a `DataReader`s data type is part of its template definition: `DataReader<Foo>`.

Using a `Subscriber` you will create a `DataReader` associating it with a specific data type, for example ‘Foo’. Note that the `Subscriber`’s `create_datareader()` method returns a generic `DataReader`. When your code is ready to access DDS data samples received for the `DataReader`, you must use type-specific operations associated with the FooDataReader, such as `read()` and `take()`.

To cast the generic `DataReader` returned by `create_datareader()` into an object of type FooDataReader, you should use the type-safe `narrow()` method of the FooDataReader class. `narrow()` will make sure that the generic `DataReader` passed to it is indeed an object of the FooDataReader class before it makes the cast. Else, it will return NULL. Figure 8.8: Simple SubscriberListener on page 502 shows an example:

```cpp
Foo_reader = FooDataReader::narrow(reader);
```

Table 8.3 DataReader Operations lists type-specific operations using a FooDataReader. Also listed are generic, non-type specific operations that can be performed using the base class object DDSDataReader (or DDS_DataReader in C). In C, you must pass a pointer to a DDS_DataReader to those generic functions.

8.4.2 Loaning and Returning Data and SampleInfo Sequences

8.4.2.1 C, Traditional C++, Java and .NET

The `read()` and `take()` operations (and their variations) return information to your application in two sequences:
• Received DDS data samples in a sequence of the data type
• Corresponding information about each DDS sample in a SampleInfo sequence

These sequences are parameters that are passed by your code into the read() and take() operations. If you use empty sequences (sequences that are initialized but have a maximum length of 0), Connext DDS will fill those sequences with memory directly loaned from the receive queue itself. There is no copying of the data or of SampleInfo when the contents of the sequences are loaned. This is certainly the most efficient way for your code to retrieve the data.

However when you do so, your code must return the loaned sequences back to Connext DDS so that they can be reused by the receive queue. If your code does not return the loan by calling the FooDataReader’s return_loan() method, then Connext DDS will eventually run out of memory to store DDS data samples received from the network for that DataReader. See Figure 8.21: Using Loaned Sequences in read() and take() below for an example of borrowing and returning loaned sequences.

```cpp
DDS_ReturnCode_t return_loan(
    FooSeq &received_data, DDS_SampleInfoSeq &info_seq);
```

Figure 8.21: Using Loaned Sequences in read() and take()

// In C++ and Java, sequences are automatically initialized
// to be empty
FooSeq data_seq;
DDS_SampleInfoSeq info_seq;
DDS_ReturnCode_t retcode;
...
// with empty sequences, a take() or read() will return loaned
// sequence elements
retcode = Foo_reader->take(data_seq, info_seq,
    DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);
    ...
// process the returned data
// must return the loaned sequences when done processing
Foo_reader->return_loan(data_seq, info_seq);
...

For the C API, you must use the FooSeq_initialize() and DDS_SampleInfoSeq_initialize() operations or the macro DDS_SEQUENCE_INITIALIZER to initialize the FooSeq and DDS_SampleInfoSeq to be empty. For example, DDS_SampleInfoSeq infoSeq; DDS_SampleInfoSeq_initialize(&infoSeq); or FooSeq fooSeq = DDS_SEQUENCE_INITIALIZER;

If your code provides its own sequences to the read/take operations, then Connext DDS will copy the data from the receive queue. In that case, you do not have to call return_loan() when you are finished with the data. However, you must make sure the following is true, or the read/take operation will fail with a return code of DDS_RETCODE_PRECONDITION_NOT_MET:
8.4.3 Accessing DDS Data Samples with Read or Take

- The received_data of type FooSeq and info_seq of type DDS_SampleInfoSeq passed in as parameters have the same maximum size (length).
- The maximum size (length) of the sequences are less than or equal to the passed in parameter, max_samples.

8.4.2.2 Modern C++

The read() and take() operations (and their variations) return LoanedSamples, an iterable collection of loaned, read-only samples each containing the actual data and meta-information about the sample. A LoanedSamples collection automatically returns the loan to the middleware in its destructor. You can also explicitly call LoanedSamples::return_loan().

Figure 8.22: Using LoanedSamples to read data

```cpp
dds::sub::LoanedSamples<Foo> samples = reader.take();
for (auto sample : samples) { // process the data
  if (sample.info().valid()) {
    std::cout << sample.data() << std::endl;
  }
}
```

8.4.3 Accessing DDS Data Samples with Read or Take

To access the DDS data samples that Connext DDS has received for aDataReader, you must invoke the read() or take() methods. These methods return a list (sequence) of DDS data samples and additional information about the DDS samples in a corresponding list (sequence) of SampleInfo structures. The contents of SampleInfo are described in 8.4.6 The SampleInfo Structure on page 565.

Calling read(), take(), or one of their variations resets the DATA_AVAILABLE status.

The way Connext DDS builds the collection of DDS samples depends on QoS policies set on the DataReader and Subscriber, the source_timestamp of the DDS samples, and the sample_states, view_states, and instance_states parameters passed to the read/take operation.

In read() and take(), you may enter parameters so that Connext DDS selectively returns DDS data samples currently stored in the DataReader’s receive queue. You may want Connext DDS to return all of the data in a single list or only a subset of the available DDS samples as configured using the sample_states, view_Connext DDStates, and instance_states masks. 8.4.6 The SampleInfo Structure on page 565 describes how these masks are used to determine which DDS data samples should be returned.

8.4.3.1 Read vs. Take

The difference between read() and take() is how Connext DDS treats the data that is returned. With take(), Connext DDS will remove the data from the DataReader’s receive queue. The data returned by Connext DDS is no longer stored by Connext DDS. With read(), Connext DDS will continue to store the data in the DataReader’s receive queue. The same data may be read again until it is taken in subsequent take()
8.4.3 Accessing DDS Data Samples with Read or Take

calls. Note that the data stored in the DataReader’s receive queue may be overwritten, even if it has not been read, depending on the setting of the 7.5.12 HISTORY QosPolicy on page 420.

The read() and take() operations are non-blocking calls, so that they may return no data (DDS_RETCODE_NO_DATA) if the receive queue is empty or has no data that matches the criteria specified by the StateMasks.

The read_w_condition() and take_w_condition() operations take a ReadCondition as a parameter instead of DDS sample, view or instance states. The only DDS samples returned will be those for which the ReadCondition is TRUE. These operations, in conjunction with ReadConditions and a WaitSet, allow you to perform ‘waiting reads.’ For more information, see 4.6.7 ReadConditions and QueryConditions on page 209.

As you will see, read and take have the same parameters:

```c
DD_ReturnCode_t read( FooSeq &received_data_seq,
                      DDS_SampleInfoSeq &info_seq,
                      DDS_Long max_samples,
                      DDS_SampleStateMask sample_states,
                      DDS_ViewStateMask view_states,
                      DDS_InstanceStateMask instance_states);

DD_ReturnCode_t take( FooSeq &received_data_seq,
                      DDS_SampleInfoSeq &info_seq,
                      DDS_Long max_samples,
                      DDS_SampleStateMask sample_states,
                      DDS_ViewStateMask view_states,
                      DDS_InstanceStateMask instance_states);
```

**Note:** These operations may loan internal Connext DDS memory, which must be returned with return_loan(). See 8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553.

Both operations return an ordered collection of DDS data samples (in the received_data_seq parameter) and information about each DDS sample (in the info_seq parameter). Exactly how they are ordered depends on the setting of the 7.4.6 PRESENTATION QosPolicy on page 362 and the 7.5.8 DESTINATION_ORDER QosPolicy on page 408. For more details please see the API Reference HTML documentation for read() and take().

In read() and take(), you can use the sample_states, view_states, and instance_states parameters to specify properties that are used to select the actual DDS samples that are returned by those methods. With different combinations of these three parameters, you can direct Connext DDS to return all DDS data samples, DDS data samples that you have not accessed before, the DDS data samples of instances that you have not seen before, DDS data samples of instances that have been disposed, etc. The possible values for the different states are described both in the API Reference HTML documentation and in 8.4.6 The SampleInfo Structure on page 565.

Table 8.16 Read and Take Operations lists the variations of the read() and take() operations.
8.4.3 Accessing DDS Data Samples with Read or Take

### Table 8.16 Read and Take Operations

<table>
<thead>
<tr>
<th>Read Operations</th>
<th>Take Operations</th>
<th>Modern C++(^1)</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>take</td>
<td>reader.read() or reader.select().state(...) .read()</td>
<td>Reads/takes a collection of DDS data samples from the DataReader. Can be used for both keyed and non-keyed data types.</td>
<td>8.4.3 Accessing DDS Data Samples with Read or Take on page 555</td>
</tr>
<tr>
<td>read_instance</td>
<td>take_instance</td>
<td>reader.select().instance(...) .read()</td>
<td>Identical to <code>read()</code> and <code>take()</code>, but all returned DDS samples belong to a single instance, which you specify as a parameter. Can only be used with keyed data types.</td>
<td>8.4.3.4 read_instance and take_instance on page 559</td>
</tr>
<tr>
<td>read_instance_w_condition</td>
<td>take_instance_w_condition</td>
<td>reader.select().instance().condition(...) .read()</td>
<td>Identical to <code>read_instance()</code> and <code>take_instance()</code>, but all returned DDS samples belong to the single specified instance and satisfy the specified ReadCondition.</td>
<td>8.4.3.7 read_instance_w_condition and take_instance_w_condition on page 561</td>
</tr>
<tr>
<td>read_next_instance</td>
<td>take_next_instance</td>
<td>reader.select().next_instance(...) .read()</td>
<td>Similar to <code>read_instance()</code> and <code>take_instance()</code>, but the actual instance is not directly specified as a parameter. Instead, the DDS samples will all belong to instance ordered after the instance that is specified by the previous_handle parameter.</td>
<td>8.4.3.5 read_next_instance and take_next_instance on page 559</td>
</tr>
<tr>
<td>read_next_instance_w_condition</td>
<td>take_next_instance_w_condition</td>
<td>reader.select().next_instance(...) .condition(...) .read()</td>
<td>Accesses a collection of DDS data samples of the next instance that match a specific set of ReadConditions, from the DataReader.</td>
<td>8.4.3.8 read_next_instance_w_condition and take_next_instance_w_condition on page 562</td>
</tr>
<tr>
<td>read_sample</td>
<td>take_sample</td>
<td>reader.select().state(DataState::not_read)</td>
<td>Provides a convenient way to access the next DDS DDS sample in the receive queue that has not been accessed before.</td>
<td>8.4.3.3 read_next_sample and take_next_sample on the next page</td>
</tr>
<tr>
<td>read_w_condition</td>
<td>take_w_condition</td>
<td>reader.select().condition(...)</td>
<td>Accesses a collection of DDS data samples from the DataReader that match specific ReadCondition criteria.</td>
<td>8.4.3.6 read_w_condition and take_w_condition on page 561</td>
</tr>
</tbody>
</table>

### 8.4.3.2 General Patterns for Accessing Data

Once the DDS data samples are available to the data readers, the DDS samples can be read or taken by the application. The basic rule is that the application may do this in any order it wishes. This approach is very flexible and allows the application ultimate control.

To access data coherently, or in order, the 7.4.6 PRESENTATION QosPolicy on page 362 must be set properly.

---

\(^1\)For the Modern C++, only the read() operation is shown; the take() variant is parallel.
**8.4.3 Accessing DDS Data Samples with Read or Take**

**Accessing DDS samples If No Order or Coherence Is Required**

Simply access the data by calling read/take on each *DataReader* in any order you want.

You do not have to call `begin_access()` and `end_access()`. However, doing so is not an error and it will have no effect.

You can call the Subscriber’s `get_datareaders()` operation to see which *DataReaders* have data to be read, but you do not need to read all of them or read them in a particular order. The `get_datareaders()` operation will return a logical 'set' in the sense that the same *DataReader* will not appear twice. The order of the *DataReaders* returned is not specified.

**Accessing DDS samples within a SubscriberListener**

This case describes how to access the data inside the listener's `on_data_on_readers()` operation (regardless of the PRESENTATION QoS policy settings).

To do so, you can call read/take on each *DataReader* in any order. You can also delegate accessing of the data to the DataReaderListeners by calling the Subscriber’s `notify_datareaders()` operation.

Similar to the previous case, you can still call the Subscriber’s `get_datareaders()` operation to determine which *DataReaders* have data to be read, but you do not have to read all of them, or read them in a particular order. `get_datareaders()` will return a logical 'set'.

You do not have to call `begin_access()` and `end_access()`. However, doing so is not an error and it will have no effect.

**8.4.3.3 read_next_sample and take_next_sample**

The `read_next_sample()` or `take_next_sample()` operation is used to retrieve the next DDS sample that hasn’t already been accessed. It is a simple way to 'read' DDS samples and frees your application from managing sequences and specifying DDS sample, instance or view states. It behaves the same as calling `read()` or `take()` with `max_samples = 1`, `sample_states = NOT_READ`, `view_states = ANY_VIEW_STATE`, and `instance_states = ANY_INSTANCE_STATE`.

```plaintext
DDS_ReturnCode_t read_next_sample(
    Foo & received_data, DDS_SampleInfo & sample_info);
DDS_ReturnCode_t take_next_sample(
    Foo & received_data, DDS_SampleInfo & sample_info);
```

It copies the next, not-previously-accessed data value from the *DataReader*. It also copies the DDS sample’s corresponding *DDS_SampleInfo* structure.

If there is no unread data in the *DataReader*, the operation will return `DDS_RETCODE_NO_DATA` and nothing is copied.

Since this operation copies both the DDS data sample and the *SampleInfo* into user-provided storage, it does not allocate nor loan memory. You do not have to call `return_loan()` after this operation.
8.4.3 Accessing DDS Data Samples with Read or Take

**Note:** If the `received_data` parameter references a structure that contains a sequence and that sequence has not been initialized, the operation will return DDS_RETCODE_ERROR.

### 8.4.3.4 read_instance and take_instance

The `read_instance()` and `take_instance()` operations are identical to `read()` and `take()`, but they are used to access DDS samples for just a specific instance (key value). The parameters are the same, except you must also supply an instance handle. These functions can only be used when the `DataReader` is tied to a keyed type, see 2.4 DDS Samples, Instances, and Keys on page 18 for more about keyed data types.

These operations may return BAD_PARAMETER if the instance handle does not correspond to an existing data-object known to the `DataReader`.

The handle to a particular data instance could have been cached from a previous `read()` operation (value taken from the `SampleInfo` struct) or created by using the `DataReader`'s `lookup_instance()` operation.

```c
DDS_ReturnCode_t read_instance(
    FooSeq &received_data,
    DDS_SampleInfoSeq &info_seq,
    DDS_Long max_samples,
    const DDS_InstanceHandle_t &a_handle,
    DDS_SampleStateMask sample_states,
    DDSViewStateMask view_states,
    DDS_InstanceStateMask instance_states);
```

**Note:** This operation may loan internal Connext DDS memory, which must be returned with `return_loan()`.

See 8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553.

### 8.4.3.5 read_next_instance and take_next_instance

The `read_next_instance()` and `take_next_instance()` operations are similar to `read_instance()` and `take_instance()` in that they return DDS samples for a specific data instance (key value). The difference is that instead of passing the handle of the data instance for which you want DDS data samples, instead you pass the handle to a ‘previous’ instance. The returned DDS samples will all belong to the 'next' instance, where the ordering of instances is explained below.

```c
DDS_ReturnCode_t read_next_instance(
    FooSeq &received_data,
    DDS_Long max_samples,
    const DDS_InstanceHandle_t &previous_handle,
    DDS_SampleStateMask sample_states,
    DDSViewStateMask view_states,
    DDS_InstanceStateMask instance_states)
```

Connext DDS orders all instances relative to each other. This ordering depends on the value of the key as defined for the data type associated with the `Topic`. For the purposes of this discussion, it is 'as if' each
instance handle is represented by a unique integer and thus different instance handles can be ordered by their value. (The ordering of the instances is specific to each implementation of the DDS standard; to maximize the portability of your code, do not assume any particular order. In the case of Connext DDS, and likely other DDS implementations, the order is not likely to be meaningful to you as a developer; it is simply important that some ordering exists.)

This operation will return values for the next instance handle that has DDS data samples stored in the receive queue (that meet the criteria specified by the StateMasks). The next instance handle will be ordered after the previous_handle that is passed in as a parameter.

The special value DDS_HANDLE_NIL can be passed in as the previous_handle. Doing so, you will receive values for the “smallest” instance handle that has DDS data samples stored in the receive queue that you have not yet accessed.

You can call the read_next_instance() operation with a previous_handle that does not correspond to an instance currently managed by the DataReader. For example, you could use this approach to iterate through all the instances, take all the DDS samples with a NOT_ALIVE_NO_WRITERS instance_state, return the loans (at which point the instance information may be removed, and thus the handle becomes invalid), and then try to read the next instance.

The example below shows how to use take_next_instance() iteratively to process all the data received for an instance, one instance at a time. We always pass in DDS_HANDLE_NIL as the value of previous_handle. Each time through the loop, we will receive DDS samples for a different instance, since the previous time through the loop, all of the DDS samples of the previous instance were returned (and thus accessed).

In the C API, you must use the FooSeq_initialize() and DDS_SampleInfoSeq_initialize() operations or the macro DDS_SEQUENCE_INITIALIZER to initialize the FooSeq and DDS_SampleInfoSeq to be empty. For example, DDS_SampleInfoSeq infoSeq; DDS_SampleInfoSeq_initialize(&infoSeq); or FooSeq fooSeq = DDS_SEQUENCE_INITIALIZER;
FooSeq received_data;
DDS_SampleInfoSeq info_seq;
while (retcode = reader->take_next_instance(received_data, info_seq,
    // the data samples returned in received_data will all
    DDS_LENGTH_UNLIMITED, DDS_HANDLE_NIL,
    DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE,
    DDS_ANY_INSTANCE_STATE)
!= DDS_RETCODE_NO_DATA) {
  // be for a single instance
  // process the data
  // now return the loaned sequences
  if (reader->return_loan(received_data, info_seq)
    != DDS_RETCODE_OK) {
    // handle error
  }
}

Note: This operation may loan internal Connext DDS memory, which must be returned with return_loan(). See 8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553.

8.4.3.6 read_w_condition and take_w_condition

The read_w_condition() and take_w_condition() operations are identical to read() and take(), but instead of passing in the sample_states, view_states, and instance_states mask parameters directly, you pass in a ReadCondition (which specifies these masks).

DDS_ReturnCode_t read_w_condition {
    FooSeq &received_data,
    DDS_SampleInfoSeq &info_seq,
    DDS_Long max_samples,
    DDSReadCondition *condition)

Note: This operation may loan internal Connext DDS memory, which must be returned with return_loan(). See 8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553.

8.4.3.7 read_instance_w_condition and take_instance_w_condition

The read_instance_w_condition() and take_instance_w_condition() operations are similar to read_instance() and take_instance(), respectively, except that the returned DDS samples must also satisfy a specified ReadCondition.

DDS_ReturnCode_t read_instance_w_condition(
    FooSeq & received_data,
    DDS_SampleInfoSeq & info_seq,
    DDS_Long max_samples,
    const DDS_InstanceHandle_t & a_handle,
    DDSReadCondition * condition);

The behavior of read_instance_w_condition() and take_instance_w_condition() follows the same rules as read() and take() regarding pre-conditions and post-conditions for the received_data and sample_info parameters.
These functions can only be used when the *DataReader* is tied to a keyed type, see 2.4 DDS Samples, Instances, and Keys on page 18 for more about keyed data types.

Similar to `read()`, these operations must be provided on the specialized class that is generated for the particular application data-type that is being accessed.

**Note:** These operations may loan internal *Connext DDS* memory, which must be returned with `return_loan()`. See 8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553.

### 8.4.3.8 `read_next_instance_w_condition` and `take_next_instance_w_condition`

The `read_next_instance_w_condition()` and `take_next_instance_w_condition()` operations are identical to `read_next_instance()` and `take_next_instance()`, but instead of passing in the sample_states, view_states, and instance_states mask parameters directly, you pass in a ReadCondition (which specifies these masks).

```c++
DDS_ReturnCode_t read_next_instance_w_condition (  
    FooSeq &received_data,  
    DDS.SampleInfoSeq &info_seq,  
    DDS_Long max_samples,  
    const DDS_InstanceHandle_t &previous_handle,  
    DDSReadCondition *condition)
```

**Note:** This operation may loan internal *Connext DDS* memory, which must be returned with `return_loan()`. See 8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553.

### 8.4.3.9 The `select()` API (Modern C++)

The Modern C++ API combines all the previous ways to read data into a single operation: `reader.select()`. This call is followed by one or more calls to functions that configure the query and always ends in a call to `read()` or `take()`. These are the functions that configure a `select()`:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>max_samples()</code></td>
<td>Specifies the maximum number of samples to read or take in this call</td>
<td>Up to the value specified in <code>max_samples_per_read</code> on page 577</td>
</tr>
<tr>
<td><code>instance()</code></td>
<td>Specifies an instance to read or take</td>
<td>All instances</td>
</tr>
<tr>
<td><code>next_instance()</code></td>
<td>Indicates that read or take should return samples for the instance that follows the one being passed (Note: both <code>next_instance()</code> and <code>instance()</code> cannot be specified at the same time)</td>
<td>All instances</td>
</tr>
<tr>
<td><code>state()</code></td>
<td>Specifies the sample state, view state and instance state</td>
<td>All samples</td>
</tr>
<tr>
<td><code>content()</code></td>
<td>Specifies a query on the data values to read</td>
<td>All samples</td>
</tr>
<tr>
<td><code>condition()</code></td>
<td>Specifies a condition (see <code>read_w_condition()</code>). If <code>condition()</code> is specified <code>state()</code> and <code>content()</code> cannot be specified. When running a query more than once on the same <em>DataReader</em>, it is more efficient to create a <code>QueryCondition</code> and pass it to <code>condition()</code> rather than using <code>content()</code></td>
<td>All samples</td>
</tr>
</tbody>
</table>
To read or take using the default options, simply call `reader.read()` or `reader.take()` with no arguments.

The following example shows how to call `select()`:

```c++
dds::sub::LoanedSamples<Foo> samples =
    reader.select()
    .max_samples(20)
    .state(dds::sub::status::DataState::new_instance())
    .content(dds::sub::Query(reader, "x > 10"))
    .instance(my_instance_handle)
    .take();
```

### 8.4.4 Acknowledging DDS Samples

DDS samples can be acknowledged one at a time, or as a group.

To explicitly acknowledge a single DDS sample:

```c++
DDS_ReturnCode_t acknowledge_sample (const DDS_SampleInfo & sample_info);
DDS_ReturnCode_t acknowledge_sample (const DDS_SampleInfo & sample_info,
                    const DDS_AckResponseData_t & response_data);
```

Or you may acknowledge all previously accessed DDS samples by calling:

```c++
DDS_ReturnCode_t DDSDataReader::acknowledge_all ()
DDS_ReturnCode_t DDSDataReader::acknowledge_all (const DDS_AckResponseData_t & response_data)
```

Where:

- `sample_info` is of type DDS_SampleInfo, identifying the DDS sample being acknowledged
- `response_data` is response data sent to the DataWriter upon acknowledgment

These operations can only be used when the `DataReader`'s 7.5.21 RELIABILITY QosPolicy on page 447 has an `acknowledgment_kind` set to DDSAPPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE. You must also set `max_app_ack_response_length` (in the 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576) to a value greater than zero.

See also: 7.3.12 Application Acknowledgment on page 317 and Guaranteed Delivery of Data (Chapter 14 on page 752).

### 8.4.5 The Sequence Data Structure

(This section doesn't apply to the Modern C++ API)

The DDS specification uses sequences whenever a variable-length array of elements must be passed through the API. This includes passing QosPolicies into Connext DDS, as well as retrieving DDS data samples from Connext DDS. A sequence is an ordered collection of elements of the same type. The type
of a sequence containing elements of type “Foo” (whether “Foo” is one of your types or a built-in Connext DDS type) is typically called “FooSeq.”

In all APIs except Java, FooSeq contains deep copies of Foo elements; in Java, which does not provide direct support for deep copy semantics, FooSeq contains references to Foo objects. In Java, sequences implement the java.util.List interface, and thus support all of the collection APIs and idioms familiar to Java programmers.

A sequence is logically composed of three things: an array of elements, a maximum number of elements that the array may contain (i.e. its allocated size), and a logical length indicating how many of the allocated elements are valid. The length may vary dynamically between 0 and the maximum (inclusive); it is not permissible to access an element at an index greater than or equal to the length.

A sequence may either “own” the memory associated with it, or it may “borrow” that memory. If a sequence owns its own memory, then the sequence itself will allocate the its memory and is permitted to grow and shrink that memory (i.e. change its maximum) dynamically.

You can also loan a sequence of memory using the sequence-specific operations loan_contiguous() or loan_discontiguous(). This is useful if you want Connext DDS to copy the received DDS data samples directly into data structures allocated in user space.

Please do not confuse (a) the user loaning memory to a sequence with (b) Connext DDS loaning internal memory from the receive queue to the user code via the read() or take() operations. For sequences of user data, these are complementary operations. read() and take() loan memory to the user, passing in a sequence that has been loaned memory with loan_contiguous() or loan_discontiguous().

A sequence with loaned of memory may not change its maximum size.

For C developers:

In C, because there is no concept of a constructor, sequences must be initialized before they are used. You can either set a sequence equal to the macro DDS_SEQUENCE_INITIALIZER or use a sequence-specific method, &lt;type&gt;Seq_initialize(), to initialize sequences.

For Traditional C++, C++/CLI, and C# developers:

Traditional C++ sequence classes overload the [ ] operators to allow you to access their elements as if the sequence were a simple array. However, for code portability reasons, Connext DDS’s Traditional C++ implementation of sequences does not use the Standard Template Library (STL).

For Java developers:

In Java, sequences implement the List interface, and typically, a List must contain Objects; it cannot contain primitive types directly. This restriction makes Lists of primitives types less efficient because each type must be wrapped and unwrapped into and from an Object as it is added to and removed from the List.
Connext DDS provides a more efficient implementation for sequences of primitive types. In Connext DDS, primitive sequence types (e.g., IntSeq, FloatSeq, etc.) are implemented as wrappers around arrays of primitive types. The wrapper also provides the usual List APIs; however, these APIs manipulate Objects that store the primitive type.

More efficient APIs are also provided that manipulate the primitive types directly and thus avoid unnecessary memory allocations and type casts. These additional methods are named according to the pattern `<standard method><primitive type>`; for example, the IntSeq class defines methods addInt() and getInt() that correspond to the List APIs add() and get(). addInt() and getInt() directly manipulate int values while add() and get() manipulate Objects that contain a single int.

For more information on sequence APIs in all languages, please consult the API Reference HTML documentation (from the main page, select Modules, RTI Connext DDS API Reference, Infrastructure Module, Sequence Support).

### 8.4.6 The SampleInfo Structure

When you invoke the read/take operations, for every DDS data sample that is returned, a corresponding SampleInfo is also returned. SampleInfo structures provide you with additional information about the DDS data samples received by Connext DDS.

Table 8.17 DDS_SampleInfo Structure shows the format of the SampleInfo structure.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_SampleStateKind</td>
<td>sample_state</td>
<td>See 8.4.6.2 Sample States on page 568</td>
</tr>
<tr>
<td>DDSViewStateKind</td>
<td>view_state</td>
<td>See 8.4.6.3 View States on page 569</td>
</tr>
<tr>
<td>DDS_InstanceStateKind</td>
<td>instance_state</td>
<td>See 8.3.8.1 Instance States on page 535</td>
</tr>
<tr>
<td>DDS_Time_t</td>
<td>source_timestamp</td>
<td>Time stored by the DataWriter when the DDS sample was written.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>instance_handle</td>
<td>Handle to the data-instance corresponding to the DDS sample.</td>
</tr>
<tr>
<td>DDS_InstanceHandle_t</td>
<td>publication_handle</td>
<td>Local handle to the DataWriter that modified the instance. This is the same instance handle returned by get_matched_publications(). You can use this handle when calling get_matched_publication_data().</td>
</tr>
</tbody>
</table>
Table 8.17 DDS_SampleInfo Structure

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>disposed_generation_count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>no_writers_generation_count</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sample_rank</td>
<td></td>
</tr>
<tr>
<td></td>
<td>generation_rank</td>
<td></td>
</tr>
<tr>
<td></td>
<td>absolute_generation_rank</td>
<td></td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>valid_data</td>
<td>Indicates whether the DDS data sample includes valid data. See 8.3.8.3 Valid Data Flag on page 539.</td>
</tr>
<tr>
<td>DDS_Time_t</td>
<td>reception_timestamp</td>
<td>Time stored when the DDS sample was committed by the DataReader. See 8.4.6.1 Reception Timestamp on page 568.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_t</td>
<td>publication_sequence_number</td>
<td>Publication sequence number assigned when the DDS sample was written by the DataWriter.</td>
</tr>
<tr>
<td>DDS_SequenceNumber_t</td>
<td>reception_sequence_number</td>
<td>Reception sequence number assigned when the DDS sample was committed by the DataReader. See 8.4.6.1 Reception Timestamp on page 568.</td>
</tr>
<tr>
<td>struct DDS_GUID_t</td>
<td>original_publication_virtual_guid</td>
<td>Original publication virtual GUID. If the Publisher’s access_scope is GROUP, this field contains the Publisher virtual GUID that uniquely identifies the DataWriter group.</td>
</tr>
<tr>
<td>struct DDS_SequenceNumber_t</td>
<td>original_publication_virtual_sequence_number</td>
<td>Original publication virtual sequence number. If the Publisher’s access_scope is GROUP, this field contains the Publisher virtual sequence number that uniquely identifies a DDS sample within the DataWriter group.</td>
</tr>
<tr>
<td>DDS_GUID_t</td>
<td>topic_query_guid</td>
<td>The GUID of the DDS_TopicQuery that is related to the sample. This GUID indicates whether a sample is part of the response to a DDS_TopicQuery or a regular (&quot;live&quot;) sample: If the sample was written for the TopicQuery stream, this field contains the GUID of the target TopicQuery. If the sample was written for the live stream, this field will be set to DDS_GUID_UNKNOWN.</td>
</tr>
</tbody>
</table>
### 8.4.6 The SampleInfo Structure

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>guid</td>
<td>Flags associated with the DDS sample; set by using the <code>flag</code> field in DDS_WriteParams_t when writing a DDS sample with <code>FooDataWriter_write_w_params()</code> (see 7.3.8 Writing Data on page 309). RTI reserves least-significant bits [0-7] for middleware-specific usage. The application can use least significant bits [8-15]. The first bit, REDELIVERED_SAMPLE, is reserved to mark a DDS sample as redelivered when using RTI Queuing Service. The second bit, INTERMEDIATE_REPLY_SEQUENCE_SAMPLE, is used to indicate that a response DDS sample is not the last response DDS sample for a given request. This bit is usually set by Connext DDS Replicators sending multiple responses for a request. The third bit, REPLICATE_SAMPLE, indicates if a sample must be broadcast by one Queuing Service replica to other replicas. The fourth bit, LAST_SHARED_READER_QUEUE_SAMPLE, indicates that a sample is the last sample in a SharedReaderQueue for a QueueConsumer DataReader. The fifth bit, INTERMEDIATE_TOPIC_QUERY_SAMPLE, indicates that a sample for a TopicQuery will be followed by more samples. This flag only applies to samples that have been published as a response to a TopicQuery. When this bit is not set and <code>topic_query_guid</code> is different from <code>GUID_UNKNOWN</code>, this sample is the last sample for that TopicQuery coming from the <code>DataWriter</code> identified by <code>original_publication_virtual_guid</code> on the previous page. The sixth bit, WRITER_REMOVED_BATCH_SAMPLE, will be set if the sample was accepted into the DataReader queue even though it was marked by the <code>DataWriter</code> as removed. Examples of removed samples in a batch are samples that were replaced due to KEEP_LAST_HISTORY_QOS on the <code>DataWriter</code> (see 7.5.12 HISTORY QosPolicy on page 420) or samples that outlived the <code>DataWriter</code>’s 7.5.14 LIFESPAN QoS Policy on page 425 duration. If the <code>DataReader</code> sets the property <code>dds.data_writer.accept_writer_removed_batch_samples</code> to true (in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439), the removed sample will be accepted into the <code>DataReader</code> queue and this flag will be set.</td>
</tr>
<tr>
<td>struct DDS_GUID_t</td>
<td>source_guid</td>
<td>The application logical data source associated with the sample.</td>
</tr>
<tr>
<td>struct DDS_GUID_t</td>
<td>related_source_guid</td>
<td>The application logical data source that is related to the sample.</td>
</tr>
<tr>
<td>struct DDS_GUID_t</td>
<td>related_subscription_guid</td>
<td>The related_reader_guid associated with the sample.</td>
</tr>
</tbody>
</table>
8.4.6 The SampleInfo Structure

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct DDS_CoherentSetInfo_t</td>
<td>coherent_set_info</td>
<td>Information about the coherent set that this sample is a part of. This field is set for all samples that are part of a coherent set. This field contains the following members:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>* group_guid identifies the DataWriter or the group of DataWriters publishing the coherent set, depending on the value of the Subscriber’s access_scope in the 7.4.6 PRESENTATION QosPolicy on page 362. (If access_scope is TOPIC or INSTANCE, then group_guid identifies a single DataWriter; if access_scope is GROUP, then group_guid identifies all the DataWriters within a Publisher.)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* coherent_set_sequence_number identifies a sample as part of a DataWriter coherent set. When the Subscriber’s access_scope in the 7.4.6 PRESENTATION QosPolicy on page 362 is TOPIC or INSTANCE, the coherent set associated with a sample is identified by the pair (group_guid, coherent_set_sequence_number).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* group_coherent_set_sequence_number identifies a sample as part of a group coherent set. When the Subscriber’s access_scope in the 7.4.6 PRESENTATION QosPolicy on page 362 is GROUP, the coherent set associated with a sample is identified by the pair (group_guid, group_coherent_set_sequence_number).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* incomplete_coherent_set indicates if a sample is part of an incomplete coherent set. An incomplete coherent set is a coherent set for which not all samples have been received. Note that a coherent set is also considered incomplete if some of its samples are filtered by content or time on the DataWriter side. By default, received samples from an incomplete coherent set are not provided to the application and they are reported as LOST_BY_INCOMPLETE_COHERENT_SET (see 8.3.7 SAMPLE_LOST Status on page 527). You can change this behavior by setting drop_incomplete_coherent_set to FALSE in the 7.4.6 PRESENTATION QosPolicy on page 362.</td>
</tr>
</tbody>
</table>

8.4.6.1 Reception Timestamp

In reliable communication, if DDS data samples are received out received of order, Connext DDS will not deliver them until all the previous DDS data samples have been received. For example, if DDS sample 2 arrives before DDS sample 1, DDS sample 2 cannot be delivered until DDS sample 1 is received. The reception_timestamp is the time when all previous DDS samples has been received—the time at which the DDS sample is committed. If DDS samples are all received in order, the committed time will be same as reception time. However, if DDS samples are lost on the wire, then the committed time will be later than the initial reception time.

8.4.6.2 Sample States

For each DDS sample received, Connext DDS keeps a sample_state relative to each DataReader. The sample_state can be either:

- **READ**: The DataReader has already accessed that DDS sample by means of read().
- **NOT_READ**: The DataReader has never accessed that DDS sample before.

The DDS samples retrieved by a read() or take() need not all have the same sample_state.
8.4.6.3 View States

For each instance (identified by a unique key value), Connext DDS keeps a view_state relative to each DataReader. The view_state can be either:

- NEW: Either this is the first time the DataReader has ever accessed DDS samples of the instance, or the DataReader has accessed previous DDS samples of the instance, but the instance has since been reborn (i.e., become not-alive and then alive again). These two cases are distinguished by examining the disposed_generation_count and the no_writers_generation_count (see 8.3.8.2 Generation Counts and Ranks on page 537).
- NOT_NEW: The DataReader has already accessed DDS samples of the same instance and the instance has not been reborn since.

The view_state in the SampleInfo structure is really a per-instance concept (as opposed to the sample_state which is per DDS sample). Thus all DDS data samples related to the same instance that are returned by read() or take() will have the same value for view_state.

8.4.6.4 Instance States

Connext DDS keeps an instance_state for each instance; it can be:

- ALIVE
- NOT_ALIVE_DISPOSED
- NOT_ALIVE_NO_WRITERS

For more information, see 8.3.8.1 Instance States on page 535.

8.4.6.5 Generation Counts and Ranks

Each DataReader keeps two counters for each new instance it detects (recall that instances are distinguished by their key values):

- disposed_generation_count
- no_writers_generation_count

For more information, see 8.3.8.2 Generation Counts and Ranks on page 537.

8.4.6.6 Valid Data Flag

The SampleInfo structure’s valid_data flag indicates whether the DDS sample contains data or is only used to communicate a change in the instance_state of the instance.
For more information, see 8.3.8.3 Valid Data Flag on page 539.

8.5 Subscriber QoS Policies

Subscribers have the same set of QoS Policies as Publishers; see 7.4 Publisher/Subscriber QoS Policies on page 345.

- 7.4.2 ENTITYFACTORY QosPolicy on page 348
- 7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) on page 350
- 7.4.4 GROUP_DATA QosPolicy on page 353
- 7.4.5 PARTITION QosPolicy on page 356
- 7.4.6 PRESENTATION QosPolicy on page 362

8.6 DataReader QoS Policies

This section describes the QoS Policies that are strictly for DataReaders (not for DataWriters). For a complete list of QoS Policies that apply to DataReaders, see Table 8.15 DataReader QoS Policies.

- 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) below
- 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576
- 8.6.3 READER_DATA_LIFECYCLE QoS Policy on page 586
- 8.6.4 TIME_BASED_FILTER QosPolicy on page 588
- 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591
- 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594

8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension)

The DATA_READER_PROTOCOL QosPolicy applies only to DataReaders that are set up for reliable operation (see 7.5.21 RELIABILITY QosPolicy on page 447). This policy allows the application to fine-tune the reliability protocol separately for each DataReader. For details of the reliable protocol used by Connext DDS, see Reliable Communications (Chapter 11 on page 688).

Connext DDS uses a standard protocol for packet (user and meta data) exchange between applications. The DataReaderProtocol QosPolicy gives you control over configurable portions of the protocol, including the configuration of the reliable data delivery mechanism of the protocol on a per DataReader basis.

These configuration parameters control timing and timeouts, and give you the ability to trade off between speed of data loss detection and repair, versus network and CPU bandwidth used to maintain reliability.
It is important to tune the reliability protocol on a per DataReader basis to meet the requirements of the end-user application so that data can be sent between DataWriters and DataReaders in an efficient and optimal manner in the presence of data loss.

You can also use this QosPolicy to control how DDS responds to "slow" reliable DataReaders or ones that disconnect or are otherwise lost.

See the 7.5.21 RELIABILITY QosPolicy on page 447 for more information on the per-DataReader/DataWriter reliability configuration. The 7.5.12 HISTORY QosPolicy on page 420 and 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 also play an important role in the DDS reliability protocol.

This policy includes the members presented in Table 8.18 DDS_DataReaderProtocolQosPolicy and Table 8.19 DDS_RtpsReliableReaderProtocol_t. For defaults and valid ranges, please refer to the API Reference HTML documentation.

When setting the fields in this policy, the following rule applies. If this is false, Connext DDS returns DDS_RETCODE_INCONSISTENT_POLICY when setting the QoS:

max_heartbeat_response_delay >= min_heartbeat_response_delay

**Table 8.18 DDS_DataReaderProtocolQosPolicy**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_GUID_t</td>
<td>virtual_guid</td>
<td>The virtual GUID (Global Unique Identifier) is used to uniquely identify the same DataReader across multiple incarnations. In other words, this value allows Connext DDS to remember information about a DataReader that may be deleted and then recreated. This value is used to provide durable reader state. For more information, see 13.2 Durability and Persistence Based on Virtual GUIDs on page 737. By default, Connext DDS will assign a virtual GUID automatically. If you want to restore the DataReader's state after a restart, you can get the DataReader's virtual GUID using its get_qos() operation, then set the virtual GUID of the restarted DataReader to the same value.</td>
</tr>
<tr>
<td>DDS_UnsignedLong</td>
<td>rtps_object_id</td>
<td>Determines the DataReader's RTPS object ID, according to the DDS-RTPS Interoperability Wire Protocol. Only the last 3 bytes are used; the most significant byte is ignored. The rtps_host_id, rtps_app_id, rtps_instance_id in the 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670, together with the 3 least significant bytes in rtps_object_id, and another byte assigned by Connext DDS identify the entity type, forms the builtinTopicId in SubscriptionBuiltInTopicData.</td>
</tr>
</tbody>
</table>
### 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension)

**Table 8.18 DDS_DataReaderProtocolQosPolicy**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>expects_inline_qos</td>
<td>Specifies whether this <em>DataReader</em> expects inline QoS with every DDS sample. <em>DataReaders</em> usually rely on the discovery process to propagate QoS changes for matched <em>DataWriters</em>. Another way to get QoS information is to have it sent inline with a DDS sample. With Connext DDS, <em>DataWriters</em> and <em>DataReaders</em> cache discovery information, so sending inline QoS is typically unnecessary. The use of inline QoS is only needed for stateless implementations of DDS in which <em>DataReaders</em> do not cache Discovery information. The complete set of QoS that a <em>DataReader</em> may send inline is specified by the Real-Time Publish-Subscribe (RTPS) Wire Interoperability Protocol. Note: The use of inline QoS creates an additional wire-payload, consuming extra bandwidth and serialization/deserialization time.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>disable_positive_acks</td>
<td>Determines whether the <em>DataReader</em> sends positive acknowledgements (ACKs) to matching <em>DataWriters</em>. When TRUE, the matching <em>DataReader</em> will keep DDS samples in its queue for this <em>DataReader</em> for a minimum keep duration (see 7.5.5.3 Disabling Positive Acknowledgements on page 396). When strict-reliability is not required and NACK-based reliability is sufficient, setting this field reduces overhead network traffic.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>propagate_dispose_of_unregistered_instances</td>
<td>Indicates whether or not an instance can move to the DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE state without being in the DDS_ALIVE_INSTANCE_STATE state. When set to TRUE, the <em>DataReader</em> will receive dispose notifications even if the instance is not alive. This field only applies to keyed <em>DataReaders</em>. To make sure the key is available to the FooDataReader's <em>get_key_value()</em> operation, use this option in combination with setting the <em>DataWriter</em>'s <em>serialize_key_with_dispose</em> field (in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389) to TRUE. See 7.5.5.5 Propagating Serialized Keys with Disposed-Instance Notifications on page 398.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>propagate_unregister_of_disposed_instances</td>
<td>Indicates whether or not an instance can move to the DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE state directly from the DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE. When set to TRUE, the <em>DataReader</em> will receive unregister notifications even if the instance is already disposed. This field only applies to keyed <em>DataReaders</em>.</td>
</tr>
</tbody>
</table>
8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension)

Table 8.18 DDS_DataReaderProtocolQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_RtpsReliableReaderProtocol_t</td>
<td>rtps_reliable_reader</td>
<td>See Table 8.19 DDS_RtpsReliableReaderProtocol_t</td>
</tr>
</tbody>
</table>

Table 8.19 DDS_RtpsReliableReaderProtocol_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>min_heartbeat_response_delay</td>
<td>Minimum delay between when the DataReader receives a heartbeat and when it sends an ACK/NACK.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>max_heartbeat_response_delay</td>
<td>Maximum delay between when the DataReader receives a heartbeat and when it sends an ACK/NACK. Increasing this value helps prevent NACK storms, but increases latency.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>heartbeat_suppression_duration</td>
<td>How long additionally received heartbeats are suppressed. When a reliable DataReader receives consecutive heartbeats within a short duration, this may trigger redundant NACKs. To prevent the DataReader from sending redundant NACKs, the DataReader may ignore the latter heartbeat(s) for this amount of time. See 11.3.4.1 How Often Heartbeats are Resent (heartbeat_period) on page 704.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>nack_period</td>
<td>Rate at which to send negative acknowledgements to new DataWriters. See 8.6.1.3 Example on page 575.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>receive_window_size</td>
<td>The number of received out-of-order DDS samples a reader can keep at a time. See 8.6.1.1 Receive Window Size on the next page</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>round_trip_time</td>
<td>The duration from sending a NACK to receiving a repair of a DDS sample. See 8.6.1.2 Round-Trip Time For Filtering Redundant NACKs on the next page</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>app_ack_period</td>
<td>The period at which application-level acknowledgment messages are sent. A DataReader sends application-level acknowledgment messages to a DataWriter at this periodic rate, and will continue sending until it receives a message from the DataWriter that it has received and processed the acknowledgment.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>min_app_ack_response_keep_duration</td>
<td>Minimum duration for which application-level acknowledgment response data is kept. The user-specified response data of an explicit application-level acknowledgment (called by DataReader's acknowledge_sample() or acknowledge_all() operations) is cached by the DataReader for the purpose of reliably resending the data with the acknowledgment message. After this duration has passed from the time of the first acknowledgment, the response data is dropped from the cache and will not be resent with future acknowledgments for the corresponding DDS sample(s).</td>
</tr>
</tbody>
</table>
Table 8.19 DDS_RtpsReliableReaderProtocol_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>samples_per_app_ack</td>
<td>The minimum number of DDS samples acknowledged by one application-level acknowledge message.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This setting applies only when the 7.5.21 RELIABILITY QosPolicy on page 447 acknowledge_kind is set to APPLICATION_EXPLICIT or APPLICATION_AUTO.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A DataReader will immediately send an application-level acknowledgment message when it has at least this many DDS samples that have been acknowledged. It will not send an acknowledgment message until it has at least this many DDS samples pending acknowledgment.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For example, calling the DataReader's acknowledge_sample() this many times consecutively will trigger the sending of an acknowledgment message. Calling the DataReader's acknowledge_all() may trigger the sending of an acknowledgment message, if at least this many DDS samples are being acknowledged at once. See 8.4.4 Acknowledging DDS Samples on page 563.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>This is independent of the DDS_RtpsReliableReaderProtocol_t's app_ack_period, where a DataReader will send acknowledgment messages at the periodic rate regardless.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When this is set to DDS_LENGTH_UNLIMITED, acknowledgment messages are sent only periodically, at the rate set by DDS_RtpsReliableReaderProtocol_t's app_ack_period.</td>
</tr>
</tbody>
</table>

8.6.1.1 Receive Window Size

A reliable DataReader presents DDS samples it receives to the user in-order. If it receives DDS samples out-of-order, it stores them internally until the other missing DDS samples are received. For example, if the DataWriter sends DDS samples 1 and 2, if the DataReader receives 2 first, it will wait until it receives 1 before passing the DDS samples to the user.

The number of out-of-order DDS samples that a DataReader can keep is set by the receive_window_size. A larger window allows more out-of-order DDS samples to be kept. When the window is full, any subsequent out-of-order DDS samples received will be rejected, and such rejections would necessitate NACK repairs that would degrade throughput. So, in network environments where out-of-order samples are more probable or where NACK repairs are costly, this window likely should be increased.

By default, the window is set to 256, which is the maximum number of DDS samples a single NACK submessage can request.

Samples rejected for exceeding the receive_window_size are counted in out_of_range_rejected_sample_count in the 8.3.7.3 DATA_READER_PROTOCOL_STATUS on page 520, but not included in the 8.3.7.8 SAMPLE_REJECTED Status on page 531.

8.6.1.2 Round-Trip Time For Filtering Redundant NACKs

When a DataReader requests for a DDS sample to be resent, there is a delay from when the NACK is sent, to when it receives the resent DDS sample. During that delay, the DataReader may receive
HEARTBEATs that normally would trigger another NACK for the same DDS sample. Such redundant repairs waste bandwidth and degrade throughput.

The **round_trip_time** is a user-configured estimate of the delay between sending a NACK to receiving a repair. A **DataReader** keeps track of when a DDS sample has been NACK’d, and will prevent subsequent NACKs from redundantly requesting for the same DDS sample, until the round trip time has passed.

Note that the default value of 0 seconds means that the **DataReader** does not filter for redundant NACKs.

### 8.6.1.3 Example

For many applications, changing these values will not be necessary. However, the more nodes that your distributed application uses, and the greater the amount of network traffic it generates, the more likely it is that you will want to consider experimenting with these values.

When a reliable **DataReader** receives a heartbeat from a **DataWriter**, it will send an ACK/NACK packet back to the **DataWriter**. Instead of sending the packet out immediately, the **DataReader** can choose to send it after a delay. This policy sets the minimum and maximum time to delay; the actual delay will be a random value in between. (For more on heartbeats and ACK/NACK messages, see Discovery (Chapter 15 on page 764).)

Why is a delay useful? For **DataWriters** that have multiple reliable **DataReaders**, an efficient way of heartbeating all of the **DataReaders** is to send a single heartbeat via multicast. In that case, all of the **DataReaders** will receive the heartbeat (approximately) simultaneously. If all **DataReaders** immediately respond with a ACK/NACK packet, the network may be flooded. While the size of a ACK/NACK packet is relatively small, as the number of **DataReaders** increases, the chance of packet collision also increases. All of these conditions may lead to dropped packets which forces the **DataWriter** to send out additional heartbeats that cause more simultaneous heartbeats to be sent, ultimately resulting in a network packet storm.

By forcing each **DataReader** to wait for a random amount of time, bounded by the minimum and maximum values in this policy, before sending an ACK/NACK response to a heartbeat, the use of the network is spread out over a period of time, decreasing the peak bandwidth required as well as the likelihood of dropped packets due to collisions. This can increase the overall performance of the reliable connection while avoiding a network storm.

When a reliable **DataReader** first matches a reliable **DataWriter**, the **DataReader** sends periodic NACK messages at the specified period to pull historical data from the **DataWriter**. The **DataReader** will stop sending periodic NACKs when it has received all historical data available at the time that it matched the **DataWriter**. The **DataReader** ensures that at least one NACK is sent per period; for example, if, within a NACK period, the **DataReader** responds to a HEARTBEAT message with a NACK, then the **DataReader** will not send another periodic NACK.

### 8.6.1.4 Properties

This QosPolicy cannot be modified after the **DataReader** is created.
It only applies to DataReaders, so there are no restrictions for setting it compatibly with respect to DataWriters.

8.6.1.5 Related QosPolicies

- 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389
- 7.5.21 RELIABILITY QosPolicy on page 447

8.6.1.6 Applicable DDS Entities

- 8.3 DataReaders on page 504

8.6.1.7 System Resource Considerations

Changing the values in this policy requires making tradeoffs between minimizing latency (decreasing min_heartbeat_response_delay), maximizing determinism (decreasing the difference between min_heartbeat_response_delay and max_heartbeat_response_delay), and minimizing network collisions/spreading out the ACK/NACK packets across a time interval (increasing the difference between min_heartbeat_response_delay and max_heartbeat_response_delay and/or shifting their values between different DataReaders).

If the values are poorly chosen with respect to the characteristics and requirements of a given application, the latency and/or throughput of the application may suffer.

8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension)

The DATA_READERRESOURCE_LIMITS QosPolicy extends your control over the memory allocated by Connext DDS for DataReaders beyond what is offered by the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451. RESOURCE_LIMITS controls memory allocation with respect to the DataReader itself: the number of DDS samples that it can store in the receive queue and the number of instances that it can manage simultaneously. DATA_READERRESOURCE_LIMITS controls memory allocation on a per matched-DataWriter basis. The two are orthogonal.

This policy includes the members in Table 8.20 DDS_DataReaderResourceLimitsQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.
### Table 8.20 DDS_DataReaderResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_remote_writers</td>
<td>Maximum number of DataWriters from which a DataReader may receive DDS data samples, among all instances. For unkeyed Topics: max_remote_writers must = max_remote_writers_per_instance</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_remote_writers_per_instance</td>
<td>Maximum number of DataWriters from which a DataReader may receive DDS data samples for a single instance. For unkeyed Topics: max_remote_writers must = max_remote_writers_per_instance</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_samples_per_remote_writer</td>
<td>Maximum number of DDS samples received out-of-order that a DataReader can store from a single reliable DataWriter. max_samples_per_remote_writer must be &lt;= RESOURCE_LIMITS::max_samples</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_infos</td>
<td>Maximum number of DDS_SampleInfo structures that a DataReader can allocate. max_infos must be &gt;= RESOURCE_LIMITS::max_samples</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_remote_writers</td>
<td>Initial number of DataWriters from which a DataReader may receive DDS data samples, including all instances. For unkeyed Topics: initial_remote_writers must = initial_remote_writers_per_instance</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_remote_writers_per_instance</td>
<td>Initial number of DataWriters from which a DataReader may receive DDS data samples for a single instance. For unkeyed Topics: initial_remote_writers must = initial_remote_writers_per_instance</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_infos</td>
<td>Initial number of DDS_SampleInfo structures that a DataReader will allocate.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_outstanding_reads</td>
<td>Initial number of times in which memory can be concurrently loaned via read/take calls without being returned with return_loan().</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_outstanding_reads</td>
<td>Maximum number of times in which memory can be concurrently loaned via read/take calls without being returned with return_loan().</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_samples_per_read</td>
<td>Maximum number of DDS samples that can be read/taken on a DataReader.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>disable_fragmentation_support</td>
<td>Determines whether the DataReader can receive fragmented DDS samples. When fragmentation support is not needed, disabling fragmentation support will save some memory resources.</td>
</tr>
</tbody>
</table>
### Table 8.20 DDS_DataReaderResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_fragmented_samples</td>
<td>The maximum number of DDS samples for which the DataReader may store fragments at a given point in time. At any given time, a DataReader may store fragments for up to max_fragmented_samples DDS samples while waiting for the remaining fragments. These DDS samples need not have consecutive sequence numbers and may have been sent by different DataWriters. Once all fragments of a DDS sample have been received, the DDS sample is treated as a regular DDS sample and becomes subject to standard QoS settings, such as max_samples. Connext DDS will drop fragments if the max_fragmented_samples limit has been reached. For best-effort communication, Connext DDS will accept a fragment for a new DDS sample, but drop the oldest fragmented DDS sample from the same remote writer. For reliable communication, Connext DDS will drop fragments for any new DDS samples until all fragments for at least one older DDS sample from that writer have been received. Only applies if disable_fragmentation_support is FALSE.</td>
</tr>
<tr>
<td></td>
<td>initial_fragmented_samples</td>
<td>The initial number of DDS samples for which a DataReader may store fragments. Only applies if disable_fragmentation_support is FALSE.</td>
</tr>
<tr>
<td></td>
<td>max_fragmented_samples_per_remote_writer</td>
<td>The maximum number of DDS samples per remote writer for which a DataReader may store fragments. This is a logical limit, so a single remote writer cannot consume all available resources. Only applies if disable_fragmentation_support is FALSE.</td>
</tr>
<tr>
<td></td>
<td>max_fragmented_samples_per_sample</td>
<td>Maximum number of fragments for a single DDS sample. Only applies if disable_fragmentation_support is FALSE.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>dynamically_allocate_fragmented_samples</td>
<td>By default, the middleware does not allocate memory upfront, but instead allocates memory from the heap upon receiving the first fragment of a new sample. The amount of memory allocated equals the amount of memory needed to store all fragments in the sample. Once all fragments of a sample have been received, the sample is deserialized and stored in the regular receive queue. At that time, the dynamically allocated memory is freed again. This QoS setting is useful for large, but variable-sized data types where up-front memory allocation for multiple samples based on the maximum possible sample size may be expensive. The main disadvantage of not pre-allocating memory is that one can no longer guarantee the middleware will have sufficient resources at run-time. If dynamically_allocate_fragmented_samples is FALSE, the middleware will allocate memory up-front for storing fragments for up to initial_fragmented_samples samples. This memory may grow up to max_fragmented_samples if needed. Only applies if disable_fragmentation_support is FALSE.</td>
</tr>
</tbody>
</table>
### Table 8.20 DDS_DataReaderResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_total_instances</td>
<td>Maximum number of instances (attached + detached instances) for which a DataReader will keep state. Only applicable if keep_min-imum_state_for_instances is TRUE. See 8.6.2.1 max_total_instances and max_instances on page 582</td>
</tr>
<tr>
<td>DDS_DataReaderResourceLimitsInstanceReplacementSettings</td>
<td>instance_replacement</td>
<td>Sets the kinds of instances allowed to be replaced for each instance state when a DataReader reaches max_instances in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451. See 8.6.2.3 Configuring DataReader Instance Replacement on page 583.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_remote_virtual_writers</td>
<td>The maximum number of virtual writers (identified by a virtual GUID) from which a DataReader may read, including all instances. When the Subscriber's access_scope is GROUP, this value determines the maximum number of DataWriter groups supported by the Subscriber. Since the Subscriber may contain more than one DataReader, only the setting of the first applies.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_remote_virtual_writers</td>
<td>The initial number of virtual writers from which a DataReader may read, including all instances.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_remote_virtual_writers_per_instance</td>
<td>Maximum number of virtual remote writers that can be associated with an instance. For unkeyed types, this value is ignored. The features of Durable Reader State and MultiChannel DataWriters, as well as Persistence Service, require Connext DDS to keep some internal state per virtual writer and instance that is used to filter duplicate DDS samples. These duplicate DDS samples could be coming from different DataWriter channels or from multiple executions of Persistence Service. Once an association between a remote virtual writer and an instance is established, it is permanent—it will not disappear even if the physical writer incarnating the virtual writer is destroyed. If max_remote_virtual_writers_per_instance is exceeded for an instance, Connext DDS will not associate this instance with new virtual writers. Duplicate DDS samples coming from these virtual writers will not be filtered on the reader. If you are not using Durable Reader State, MultiChannel DataWriters or Persistence Service, you can set this property to 1 to optimize resources. For additional information about the virtual writers see Mechanisms for Achieving Information Durability and Persistence (Chapter 13 on page 732).</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_remote_virtual_writers_per_instance</td>
<td>Initial number of virtual remote writers per instance. For unkeyed types, this value is ignored.</td>
</tr>
</tbody>
</table>
### Table 8.20 DDS_DataReaderResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_remotewriters_per_sample</td>
<td>Maximum number of remote writers that are allowed to write the same DDS sample. One scenario in which two DataWriters may write the same DDS sample is when using Persistence Service. The DataReader may receive the same DDS sample from the original DataWriter and from an Persistence Service DataWriter.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_query_condition_filters</td>
<td>This value determines the maximum number of unique query condition content filters that a reader may create. Each query condition content filter is comprised of both its query_expression and query_parameters. Two query conditions that have the same query_expression will require unique query condition filters if their query_parameters differ. Query conditions that differ only in their state masks will share the same query condition filter.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_app_ack_response_length</td>
<td>Maximum length of application-level acknowledgment response data. The maximum length of response data in an application-level acknowledgment. When set to zero, no response data is sent with application-level acknowledgments.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>keep_minimum_state_for_instances</td>
<td>Determines whether the DataReader keeps a minimum instance state for up to max_total_instances. The minimum state is useful for filtering samples in certain scenarios. See 8.6.2.1 max_total_instances and max_instances on page 582</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_topic_queries</td>
<td>The initial number of TopicQueries allocated by a DataReader.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_topic_queries</td>
<td>The maximum number of active TopicQueries that a DataReader can create. Once this limit is reached, a DataReader can create more TopicQueries only if it deletes some of the previously created ones.</td>
</tr>
<tr>
<td>DDS_AllocationSettings_t</td>
<td>shmem_ref_transfer_mode_attached_segment_allocation</td>
<td>Configures the allocation resource used to attach to different shared memory segments if you are using Zero Copy transfer over shared memory. See 23.1.5 Zero Copy Transfer Over Shared Memory on page 962.</td>
</tr>
</tbody>
</table>

**DataReaders** must allocate internal structures to handle: the maximum number of **DataWriters** that may connect to it; whether or not a **DataReader** handles data fragmentation and how many data fragments that it may handle (for **DDS** data samples larger than the MTU of the underlying network transport); how many simultaneous outstanding loans of internal memory holding DDS data samples can be provided to user code; as well as others.

Most of these internal structures start at an initial size and, by default, will grow as needed by dynamically allocating additional memory. You may set fixed, maximum sizes for these internal structures if you want
initial size to the maximum size will prevent Connext DDS from dynamically allocating any memory after the DataReader is created.

This policy also controls how the allocated internal data structure may be used. For example, DataReaders need data structures to keep track of all of the DataWriters that may be sending it DDS data samples. The total number of DataWriters that it can keep track of is set by the initial_remote_writers and max_remote_writers values. For keyed Topics, initial_remote_writers_per_instance and max_remote_writers_per_instance control the number of DataWriters allowed by the DataReader to modify the value of a single instance.

By setting the max value to be less than max_remote_writers, you can prevent instances with many DataWriters from using up the resources and starving other instances. Once the resources for keeping track of DataWriters are used up, the DataReader will not be able to accept “connections” from new DataWriters. The DataReader will not be able to receive data from new matching DataWriters which would be ignored.

In the reliable protocol used by Connext DDS to support a RELIABLE setting for the 7.5.21 RELIABILITY QosPolicy on page 447, the DataReader must temporarily store DDS data samples that have been received out-of-order from a reliable DataWriter. The storage of out-of-order DDS samples is allocated from the DataReader’s receive queue and shared among all reliable DataWriters. The parameter max_samples_per_remote_writer controls the maximum number of out-of-order data DDS samples that the DataReader is allowed to store for a single DataWriter. This value must be less than the max_samples value set in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.

max_samples_per_remote_writer allows Connext DDS to share the limited resources of the DataReader equitably so that a single DataWriter is unable to use up all of the storage of the DataReader while missing DDS data samples are being resent.

When setting the values of the members, the following rules apply:

- \( \text{max\_remote\_writers} \geq \text{initial\_remote\_writers} \)
- \( \text{max\_remote\_writers\_per\_instance} \geq \text{initial\_remote\_writers\_per\_instance} \)
  \( \text{max\_remote\_writers\_per\_instance} \leq \text{max\_remote\_writers} \)
- \( \text{max\_infos} \geq \text{initial\_infos} \)
  \( \text{max\_infos} \geq \text{RESOURCE\_LIMITS\::max\_samples} \)
- \( \text{max\_outstanding\_reads} \geq \text{initial\_outstanding\_reads} \)
- \( \text{max\_remote\_writers} \geq \text{max\_remote\_writers\_per\_instance} \)
- \( \text{max\_samples\_per\_remote\_writer} \leq \text{RESOURCE\_LIMITS\::max\_samples} \)

If any of the above are false, Connext DDS returns the error code DDS_RETCODE_INCONSISTENT_POLICY when setting the DataReader’s QoS.
8.6.2.1 max_total_instances and max_instances

The features 13.4 Durable Reader State on page 744, Multi-channel DataWriters (Chapter 20 on page 896), and Persistence Service (Part 8: RTI Persistence Service on page 1179) require Connext DDS to keep some minimum internal state even for instances without DataWriters or DDS samples in the DataReader’s queue or that have been purged due to a dispose. Instances for which only this minimum state is kept are called detached instances. The additional state is used to filter duplicate DDS samples that could be coming from different DataWriter channels or from multiple executions of Persistence Service. The total maximum number of instances that will be managed by the middleware, attached plus detached instances, is determined by max_total_instances. This additional state will only be kept for up to max_total_instances if keep_minimum_state_for_instances is TRUE, otherwise the additional state will not be kept for any instances. The minimum state includes information such as the source timestamp of the last sample received by the instance and the last sequence number received from a virtual GUID. See also 8.3.8.6.1 Active State and Minimum State on page 540.

8.6.2.2 keep_minimum_state_for_instances

There are important implications of the minimum state setting.

When a DataReader is exposed to an unbounded number of instances over its lifetime (for example, if the key for an instance is a UUID and the application cycles through unlimited numbers of such UUIDs over time) and the DataReader does keep its minimum state, the set of minimum state data will grow with the total number of instances (unique keys) the DataReader has been exposed to until max_total_instances is reached.

max_total_instances by default gets its value from max_instances. If max_instances is set to its default value, which is unbounded, the DataReader’s memory will grow slowly but without bound until the DataReader itself is deleted. As a rule of thumb, when instances are used only once in a system and are never used again after being disposed or unregistered, set max_instances and max_total_instances to finite values or bound the lifetime of the DataReader (see 4.1.1 Creating and Deleting DDS Entities on page 167). If neither of these options is practical, it may help to set keep_minimum_state_for_instances to FALSE.

If a DataReader does not retain this minimum state, there may be correctness implications if the DataReader is exposed to an instance again after it has been removed from the DataReader cache. For example, because the last source timestamp is not preserved, eventual consistency cannot be assured (even if destination order is by source timestamp). Samples that had already been received by the DataReader may be re-delivered and provided to the application again as if for the first time (especially when using redundant Routing Service routes, Persistence Service, or Collaborative DataWriters). As a rule of thumb, when instances have complex lifecycles (especially involving multiple DataWriters modifying the instance), in which an instance can become not alive and later come alive again, set keep_minimum_state_for_instances to TRUE.
8.6.2.3 Configuring DataReader Instance Replacement

When the max_instances limit in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 is reached, a DataReader will try to make space for a new instance by replacing an existing instance according to the instance replacement kind set in instance_replacement in the 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576. If it cannot make space for the new instance, the sample for the new instance will be lost with the reason LOST_BY_INSTANCES_LIMIT (see 8.3.7.7 SAMPLE LOST Status on page 527).

The instance_replacement field is useful for managing large volumes of instances that come and go. It is important to be able to set an upper limit on the resources that will be used by an application to avoid running into decreased performance and potentially running out of system resources. The instance replacement QoS setting allows you to set an upper bound on the resources that will be used for instances. It allows DataReaders to make room for new instances by replacing older ones. For example, a hospital may have 100 beds. Many patients (instances) come and go, so at any given time you only need resources for 100 instances, but over time you will see an unbounded number of instances. An instance replacement policy can help manage this flow.

For each instance state (see 8.3.8.1 Instance States on page 535), you can set the following removal kinds:

- The alive_instance_removal kind sets a removal policy for ALIVE instances (default: DDS_NO_INSTANCE_REMOVAL).

- The disposed_instance_removal kind sets a removal policy for NOT_ALIVE_DISPOSED instances (default: DDS_EMPTY_INSTANCE_REMOVAL).

- The no_writers_instance_removal kind sets a removal policy for NOT_ALIVE_NO_WRITERS instances (default: DDS_EMPTY_INSTANCE_REMOVAL).

For each instance state, you can choose among the following replacement kinds:

- DDS_NO_INSTANCE_REMOVAL: Instances in the associated state cannot be replaced.

- DDS_EMPTY_INSTANCE_REMOVAL: Instances in the associated state can be replaced only if they are empty (all samples have been taken or removed from the DataReader queue due to QoS settings such as, but not limited to, the 7.5.14 LIFESPAN QoS Policy on page 425 or sample purging due to the 8.6.3 READER_DATA_LIFECYCLE QoS Policy on page 586), and there are no outstanding loans on any of the instance's samples.

- DDS_FULLY PROCESSED_INSTANCE_REMOVAL: Instances in the associated state can be replaced only if every sample has been processed by the application. A sample is considered processed by the application based on the Reliability kind:
  - If the Reliability kind is RELIABLE, a sample is considered processed by the application based on the ApplicationAcknowledgementKind (see 7.3.12.1 Application...
Acknowledgment Kinds on page 318):

- **PROTOCOL_ACKNOWLEDGMENT_MODE or APPLICATION_AUTO_ACKNOWLEDGEMENT_MODE**: The sample is considered processed when it has been read or taken by the application and return_loan has been called.

- **APPLICATION_EXPLICIT_ACKNOWLEDGEMENT_MODE**: The sample is considered processed when the subscribing application has explicitly acknowledged the sample by calling either the DataReader’s acknowledge_sample() or acknowledge_all() operations, the AppAckConf message has been received, and the application has called return_loan.

- If the Reliability kind is BEST EFFORT, a sample is considered processed by the application when all samples have been read or taken by the application and return_loan has been called.

- **DDS_ANY_INSTANCE_REMOVAL**: Instances in the associated state can be replaced regardless of whether the subscribing application has processed all of the samples. Samples that have not been processed will be dropped and accounted for by the total_samples_dropped_by_instance_replacement statistic in the 8.3.7.2 DATA_READER_CACHE_STATUS on page 517.

For all kinds, instance replacement starts with the least-recently-updated (LRU) instance that matches the allowed criteria. For example, if alive_instance_removal is set to DDS_EMPTY_INSTANCE_REMOVAL: when the max_instances limit is reached, the least-recently-updated, empty, ALIVE instance will be replaced to make room for the new instance. An instance is considered updated when a valid sample or dispose sample for the instance is received and accepted by the DataReader. An instance is not considered updated in the following cases:

- When using EXCLUSIVE_OWNERSHIP, when samples that are received from DataWriters that do not own the instance. Only the owner of an instance can update the instance.

- A sample that is filtered out due to content filtering does not count as updating the instance.

- Unregister messages do not count as an update to the instance because the unregister message conveys information about the DataWriter (that it is finished updating the instance), as opposed to any change to the instance itself.

There is no preference among the instance states as far as which instance is replaced first; instance replacement relies only on the LRU. For example, imagine if Connext DDS were to prefer disposed_instance_removal over alive_instance_removal. It doesn't, but if it did, the application might never see disposed instances, yet have very old alive instances in its queue. The same is true for the replacement criteria options. If you choose DDS_FULLY_PROCESSED_INSTANCE_REMOVAL (for example), Connext DDS will not look for empty instances first and then fully processed instances; the LRU instance that is considered fully-processed will be replaced.
If no replaceable instance exists after the instance replacement kinds above have been applied, the sample for the new instance will be considered lost with the reason LOST_BY_INSTANCES_LIMIT in the 8.3.7.7 SAMPLE_LOST Status on page 527; the instance will not be inserted into the DataReader queue.

### 8.6.2.4 Example

The `max_samples_per_remote_writer` value affects sharing and starvation. `max_samples_per_remote_writer` can be set to less than the RESOURCE_LIMITS QosPolicy’s `max_samples` to prevent a single `DataWriter` from starving others. This control is especially important for `Topics` that have their 7.5.17 OWNERSHIP QosPolicy on page 434 set to SHARED.

In the case of EXCLUSIVE ownership, a lower-strength remote `DataWriter` can "starve" a higher-strength remote `DataWriter` by making use of more of the `DataReader`'s resources, an undesirable condition. In the case of SHARED ownership, a remote `DataWriter` may starve another remote `DataWriter`, making the sharing not really equal.

### 8.6.2.5 Properties

This QosPolicy cannot be modified after the `DataReader` is created.

It only applies to `DataReaders`, so there are no restrictions for setting it compatibly on the `DataWriter`.

### 8.6.2.6 Related QosPolicies

- 7.5.22 RESOURCE_LIMITS QosPolicy on page 451
- 7.5.17 OWNERSHIP QosPolicy on page 434

### 8.6.2.7 Applicable DDS Entities

- 8.3 DataReaders on page 504

### 8.6.2.8 System Resource Considerations

Increasing any of the “initial” values in this policy will increase the amount of memory allocated by Connext DDS when a new `DataReader` is created. Increasing any of the “max” values will not affect the initial memory allocated for a new `DataReader`, but will affect how much additional memory may be allocated as needed over the `DataReader`’s lifetime.

Setting a max value greater than an initial value thus allows your application to use memory more dynamically and efficiently in the event that the size of the application is not well-known ahead of time. However, Connext DDS may dynamically allocate memory in response to network communications.
8.6.3 READER_DATA_LIFECYCLE QoS Policy

This policy controls the behavior of the DataReader with regards to the lifecycle of the data instances it manages, that is, the data instances that have been received and for which the DataReader maintains some internal resources.

When a DataReader receives data, it is stored in a receive queue for the DataReader. The user application may either take the data from the queue or leave it there. This QoS controls whether or not Connext DDS will automatically remove data from the receive queue (so that user applications cannot access it afterward) when Connext DDS detects that there are no more DataWriters alive for that data.

DataWriters may also call dispose() on its data, informing DataReaders that the data no longer exists. This QosPolicy also controls whether or not Connext DDS automatically removes disposed data from the receive queue.

For keyed Topics, the consideration of removing DDS data samples from the receive queue is done on a per instance (key) basis. Thus when Connext DDS detects that there are no longer DataWriters alive for a certain key value for a Topic (an instance of the Topic), it can be configured to remove all DDS data samples for a certain instance (key). DataWriters also can dispose its data on a per instance basis. Only the DDS data samples of disposed instances would be removed by Connext DDS if so configured.

This policy helps purge untaken DDS samples from not-alive-instances and thus may prevent a DataReader from reclaiming resources. With this policy, the untaken DDS samples from not-alive-instances are purged and treated as if the DDS samples were taken after the specified amount of time.

The DataReader internally maintains the DDS samples that have not been taken by the application, subject to the constraints imposed by other QoS policies such as 7.5.12 HISTORY QosPolicy on page 420 and 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.

The DataReader also maintains information regarding the identity, view-state, and instance-state of data instances, even after all DDS samples have been ‘taken’ (see 8.4.3 Accessing DDS Data Samples with Read or Take on page 555). This is needed to properly compute the states when future DDS samples arrive.

Under normal circumstances, a DataReader can only reclaim all resources for instances for which there are no DataWriters and for which all DDS samples have been ‘taken.’ The last DDS sample taken by the DataReader for that instance will have an instance state of NOT_ALIVE_NO_WRITERS or NOT_ALIVE_DISPOSED_INSTANCE (depending on whether or not the instance was disposed by the last>DataWriter that owned it.) If you are using the default (infinite) values for this QosPolicy, this behavior can cause problems if the application does not ‘take’ those DDS samples for some reason. The ‘untaken’ DDS samples will prevent the DataReader from reclaiming the resources and they would remain in the DataReader indefinitely.

A DataReader can also reclaim all resources for instances that have an instance state of NOT_ALIVE_DISPOSED and for which all DDS samples have been ‘taken’. DataReaders will only reclaim resources in this situation when autopurge_disposed_instances_delay has been set to zero.
It includes the members in Table 8.21 DDS_ReaderDataLifecycleQosPolicy.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>autopurge_nowriter_samples_delay</td>
<td>Minimum duration for which the DataReader will maintain samples regarding an instance once its instance_state becomes NOT_ALIVE_NO_WRITERS. An instance will transition to NOT_ALIVE_NO_WRITERS when all known writers for the instance have lost liveliness, been deleted, or unregistered from the instance. After this time elapses, the DataReader will purge all samples for the instance even if they have not been read by the application. These samples will be dropped. (See expired_dropped_sample_count in 8.3.7.2 DATA_READER_CACHE_STATUS on page 517.) This purge is done lazily when space is needed for other samples or instances (for example, when a resource limit such as max_samples on page 452 is hit). Default: INFINITE.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>autopurge_disposed_samples_delay</td>
<td>Minimum duration for which the DataReader will maintain samples for an instance once its instance_state becomes NOT_ALIVE_DISPOSED. After this time elapses, the DataReader will purge all samples for the instance even if they have not been read by the application. These samples will be dropped. (See expired_dropped_sample_count in 8.3.7.2 DATA_READER_CACHE_STATUS on page 517.) This purge is done lazily when space is needed for other samples or instances (for example, when a resource limit such as max_samples on page 452 is hit). Default: INFINITE</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>autopurge_disposed_instances_delay</td>
<td>Minimum duration for which the DataReader will maintain &quot;active state&quot; information about a received instance once its instance_state becomes NOT_ALIVE_DISPOSED, and there are no samples for the instance in the DataReader queue. (See 8.3.8.6.1 Active State and Minimum State on page 540.) Note: only values of 0 or INFINITE are currently supported. A value of 0 will purge an instance's state immediately after the instance state transitions to NOT_ALIVE_DISPOSED, as long as all samples, including the dispose sample, associated with that instance have been 'taken.' After this time elapses, when the last sample for the disposed instance is taken, the DataReader will keep only a minimum amount of state about the instance. To disable retention of even this minimum state after the delay period, also set keep_minimum_state_for_instances to FALSE in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576. See 8.6.2.2 keep_minimum_state_for_instances on page 582. Default: INFINITE</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>autopurge_nowriter_instances_delay</td>
<td>Minimum duration for which the DataReader will maintain &quot;active state&quot; information about a received instance once its instance_state becomes NOT_ALIVE_NO_WRITERS and there are no samples for the instance in the DataReader queue. (See 8.3.8.6.1 Active State and Minimum State on page 540.) An instance will transition to NOT_ALIVE_NO_WRITERS when all known writers for the instance have lost liveliness, been deleted, or unregistered from the instance. After this time elapses, when the last sample for the instance without writers is taken, the DataReader will keep only the minimum state about the instance. To disable retention of even this minimum state after the delay period, also set keep_minimum_state_for_instances to FALSE in the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576. See 8.6.2.2 keep_minimum_state_for_instances on page 582. (Note: only values of 0 or INFINITE are currently supported. A value of 0 will purge an instance's state immediately after the instance state transitions to NOT_ALIVE_NO_WRITERS, as long as all samples, including the no_writers sample, associated with that instance have been 'taken.') Default: 0</td>
</tr>
</tbody>
</table>

### 8.6.3.1 Properties

This QoS policy can be modified after the DataReader is enabled.
8.6.4 TIME_BASED_FILTER QosPolicy

It only applies to DataReaders, so there are no RxO restrictions for setting it compatibly on the DataWriter.

8.6.3.2 Related QoS Policies

- 7.5.12 HISTORY QosPolicy on page 420
- 7.5.15 LIVELINESS QosPolicy on page 427
- 7.5.17 OWNERSHIP QosPolicy on page 434
- 7.5.22 RESOURCE_LIMITS QosPolicy on page 451
- 7.5.31 WRITER_DATA_LIFECYCLE QoS Policy on page 467

8.6.3.3 Applicable DDS Entities

- 8.3 DataReaders on page 504

8.6.3.4 System Resource Considerations

None.

8.6.4 TIME_BASED_FILTER QosPolicy

The TIME_BASED_FILTER QosPolicy allows you to specify that data should not be delivered more than once per specified period for data-instances of a DataReader—regardless of how fast DataWriters are publishing new DDS samples of the data-instance.

This QoS policy allows you to optimize resource usage (CPU and possibly network bandwidth) by only delivering the required amount of data to different DataReaders.

DataWriters may send data faster than needed by a DataReader. For example, a DataReader of sensor data that is displayed to a human operator in a GUI application does not need to receive data updates faster than a user can reasonably perceive changes in data values. This is often measure in tenths (0.1) of a second up to several seconds. However, a DataWriter of sensor information may have DataReaders that are processing the sensor information to control parts of the system and thus need new data updates in measures of hundredths (0.01) or thousandths (0.001) of a second.

With this QoS policy, different DataReaders can set their own time-based filters, so that data published faster than the period set by a DataReader will be dropped by the middleware and not delivered to the DataReader. Note that all filtering takes place on the reader side.

It includes the member in Table 8.22 DDS_TimeBasedFilterQosPolicy. For the default and valid range, please refer to the API Reference HTML documentation.
Table 8.22 DDS_TimeBasedFilterQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>minimum_separation</td>
<td>Minimum separation time between DDS samples of the same instance. Must be &lt;= DEADLINE::period</td>
</tr>
</tbody>
</table>

As seen in Figure 8.23: Accepting Data for DataReaders below, it is inconsistent to set a DataReader’s minimum_separation longer than its 7.5.7 DEADLINE QosPolicy on page 405 period.

Figure 8.23: Accepting Data for DataReaders

DDS data samples for a DataReader can be filtered out using the TIME_BASED_FILTER QoS (minimum_separation). Once a DDS sample for an instance has been received, Connext DDS will accept but drop any new data samples for the same instance that arrives within the time specified by minimum_separation. After the minimum_separation, a new DDS sample that arrives is accepted and stored in the receive queue, and the timer starts again. If no DDS samples arrive by the DEADLINE, the REQUESTED_DEADLINE_MISSED status will be changed and Listeners called back if installed.

This QosPolicy allows a DataReader to subsample the data being published for a data instance by DataWriters. If a user application only needs new DDS samples for a data instance to be received at a specified period, then there is no need for Connext DDS to deliver data faster than that period. However, whether or not data being published by a DataWriter at a faster rate than set by the TIME_BASED_FILTER QoS is sent on the wire depends on several factors, including whether the DataReader is receiving the data reliably and if the data is being sent via multicast for multiple DataReaders.

For best effort data delivery, if the data type is unkeyed and the DataWriter has an infinite liveliness lease_duration (7.5.15 LIVELINESS QosPolicy on page 427), Connext DDS will only send as many packets to a DataReader as required by the TIME_BASED_FILTER, no matter how fast the DataWriter’s write() function is called.
For multicast data delivery to multiple DataReaders, the DataReader with the lowest TIME_BASED_FILTER minimum_separation determines the DataWriter's send rate. For example, if a DataWriter sends multicast to two DataReaders, one with minimum_separation of 2 seconds and one with minimum_separation of 1 second, the DataWriter will send every 1 second.

Other configurations (for example, when the DataWriter is reliable, or the data type is keyed, or the DataWriter has a finite liveliness lease_duration) must send all data published by the DataWriter. On reception, only the data that passes the TIME_BASED_FILTER will be stored in the DataReader’s receive queue. Extra data will be accepted but dropped. Note that filtering is only applied on ‘alive’ DDS samples (that is, DDS samples that have not been disposed/unregistered).

### 8.6.4.1 Example

The purpose of this QosPolicy is to prevent fast DataWriters from overwhelming a DataReader that cannot process the data at the rate the data is being published. In certain configurations, the number of packets sent by Connext DDS can also be reduced thus minimizing the consumption of network bandwidth.

You may want to change the minimum_separation between DDS data samples for one or more of the following reasons:

- The DataReader is connected to the network via a low-bandwidth connection that is unable to sustain the amount of traffic generated by the matched DataWriter(s).

- The rate at which the matched DataWriter(s) can generate DDS samples is faster than the rate at which the DataReader can process them. Or faster than needed by the DataReader. For example, a graphical user interface seldom needs to be updated faster than 30 times a second, even if new data values are available much faster.

- The resource limits of the DataReader are constrained relative to the number of DDS samples that could be generated by the matched DataWriter(s). Too many packets coming at once will cause them to be exhausted before the DataReader has time to process them.

### 8.6.4.2 Properties

This QosPolicy can be modified at any time.

It only applies to DataReaders, so there are no restrictions for setting it compatibly on the DataWriter.

### 8.6.4.3 Related QosPolicies

- 7.5.21 RELIABILITY QosPolicy on page 447
- 7.5.7 DEADLINE QosPolicy on page 405
- 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on the next page
8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension)

8.6.4.4 Applicable DDS Entities

- 8.3 DataReaders on page 504

8.6.4.5 System Resource Considerations

Depending on the values of other QosPolicies such as RELIABILITY and TRANSPORT_MULTICAST, this policy may be able to decrease the usage of network bandwidth and CPU by preventing unneeded packets from being sent and processed.

8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension)

This QosPolicy specifies the multicast address on which aDataReader wants to receive its data. It can also specify a port number as well as a subset of the available transports with which to receive the multicast data.

By default, DataWriters will send individually addressed packets for each DataReader that subscribes to the topic of the DataWriter—this is known as unicast delivery. Thus, as many copies of the data will be sent over the network as there are DataReaders for the data. The network bandwidth used by a DataWriter will thus increase linearly with the number of DataReaders.

Multicast is a concept supported by some transports, most notably UDP/IP, so that a single packet on the network can be addressed such that it is received by multiple nodes. This is more efficient when the same data needs to be sent to multiple nodes. By using multicast, the network bandwidth usage will be constant, independent of the number of DataReaders.

Coordinating the multicast address specified by DataReaders can help optimize network bandwidth usage in systems where there are multiple DataReaders for the same Topic.

The QosPolicy structure includes the members in Table 8.23 DDS_TransportMulticastQosPolicy.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TransportMulticastSettingSeq (A sequence of the type shown in Table 8.24 DDS_TransportMulticastSetting_t)</td>
<td>value</td>
<td>A sequence of up to 16 multicast locators. This is a hard limit that cannot be increased. However, this limit can be decreased by configuring the DomainParticipant property dds.domain Participant.max_announced_locator_list_size. For more information on the locator format, see 15.2.1.1 Locator Format on page 770.</td>
</tr>
</tbody>
</table>
Table 8.23 DDS_TransportMulticastQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TransportMulticastKind</td>
<td>kind</td>
<td>This field can be set to one of the following two values: DDS AUTOMATIC TRANSPORT MULTICAST QOS or DDS UNICAST ONLY TRANSPORT MULTICAST QOS. If it is set to DDS AUTOMATIC TRANSPORT MULTICAST QOS, the behavior depends on the content of DDS_TransportMulticastQosPolicy::value: If DDS_TransportMulticastQosPolicy::value does not have any elements, multicast will not be used. If DDS_TransportMulticastQosPolicy::value first element has an empty address, the address will be obtained from DDS_TransportMulticastMappingQosPolicy. If none of the elements in DDS_TransportMulticastQosPolicy::value are empty, and at least one element has a valid address, then that address will be used. If it is set to DDS UNICAST ONLY TRANSPORT MULTICAST QOS, then multicast will not be used.</td>
</tr>
</tbody>
</table>

Table 8.24 DDS_TransportMulticastSetting_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_StringSeq</td>
<td>transports</td>
<td>A sequence of transport aliases that specifies which transports should be used to receive multicast messages for this DataReader.</td>
</tr>
<tr>
<td>char*</td>
<td>receive_address</td>
<td>A multicast group address to which the DataWriter should send data for this DataReader.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>receive_port</td>
<td>The port that should be used in the addressing of multicast messages destined for this DataReader. A value of 0 will cause Connext DDS to use a default port number based on domain ID. See 15.5 Ports Used for Discovery on page 791.</td>
</tr>
</tbody>
</table>

To take advantage of multicast, the value of this QosPolicy must be coordinated among all of the applications on a network for DataReaders of the same Topic. For a DataWriter to send a single packet that will be received by all DataReaders simultaneously, the same multicast address must be used.

To use this QosPolicy, you will also need to specify a port number. A port number of 0 will cause Connext DDS to automatically use a default value. As explained in 15.5 Ports Used for Discovery on page 791, the default port number for multicast addresses is based on the domain ID. Should you choose to use a different port number, then for every unique port number used by Entities in your application, depending on the transport, Connext DDS may create a thread to process messages received for that port on that transport. See Connext DDS Threading Model (Chapter 21 on page 908) for more about threads.

Threads are created on a per-transport basis, so if this QosPolicy specifies multiple transports for a receive_port, then a thread may be created for each transport for that unique port. Some transports may be able to share a single thread for different ports, others can not. Note that different Entities can share the same port number, and thus, the same thread will process all of the data for all of the Entities sharing the same port number for a transport.
Also note that if the port number specified by this QoS is the same as a port number specified by a TRANSPORT_UNICAST QoS, then the transport may choose to process data received both via multicast and unicast with a single thread. Whether or not a transport must use different threads to process data received via multicast or unicast for the same port number depends on the implementation of the transport.

Notes:

- The same multicast address can be used by DataReaders of different Topics.
- Even though the TRANSPORT_MULTICAST QoS allows you to specify multiple multicast addresses for a DataReader, Connext DDS currently only uses one multicast address (the first in the sequence) per DataReader.
- If a DataWriter is using the 7.5.16 MULTICHANNEL QosPolicy (DDS Extension) on page 432, the multicast addresses specified in the TRANSPORT_MULTICAST QosPolicy are ignored by that DataWriter. The DataWriter will not publish DDS samples on those locators.

8.6.5.1 Example

In an airport, there may be many different monitors that display current flight information. Assuming each monitor is controlled by a networked application, network bandwidth would be greatly reduced if flight information was published using multicast.

Figure 8.24: Setting Up a Multicast DataReader below shows an example of how to set this QosPolicy.

Figure 8.24: Setting Up a Multicast DataReader

```java
... 
DDS_DataReaderQos reader_qos;
reader_listener = new HelloWorldListener();
if (reader_listener == NULL) {
    // handle error
}
// Get default data reader QoS to customize
retcode = subscriber->get_default_datareader_qos(reader_qos);
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Set up multicast reader
reader_qos.multicast.value.ensure_length(1,1);
reader_qos.multicast.value[0].receive_address =
    DDS_String_dup("239.192.0.1");
reader = subscriber->create_datareader(
    topic,reader_qos,
    reader_listener, DDS_STATUS_MASK_ALL);
```

8.6.5.2 Properties

This QosPolicy cannot be modified after the Entity is created.
For compatibility between DataWriters and DataReaders, the DataWriter must be able to send to the multicast address that the DataReader has specified.

### 8.6.5.3 Related QosPolicies

- 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432
- 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460
- 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665

### 8.6.5.4 Applicable DDS Entities

- 9.3 DomainParticipants on page 609
- 8.3 DataReaders on page 504

### 8.6.5.5 System Resource Considerations

On Ethernet-based systems, the number of multicast addresses that can be “listened” to by the network interface card is usually limited. The exact number of multicast addresses that can be monitored simultaneously by a NIC depends on its manufacturer. Setting a multicast address for a DataReader will use up one of the multicast-address slots of the NIC.

What happens if the number of different multicast addresses used by different DataReaders across different applications on the same node exceeds the total number supported by a NIC depends on the specific operating system. Some will prevent you from configuring too many multicast addresses to be monitored.

Many operating systems will accommodate the extra multicast addresses by putting the NIC in promiscuous mode. This means that the NIC will pass every Ethernet packet to the operating system, and the operating system will pass the packets with the specified multicast addresses to the application(s). This results in extra CPU usage. We recommend that your applications do not use more multicast addresses on a single node than the NICs on that node can listen to simultaneously in hardware.

Depending on the implementation of a transport, Connext DDS may need to create threads to receive and process data on a unique-port-number basis. Some transports can share the same thread to process data received for different ports; others like UDPv4 must have different threads for different ports. In addition, if the same port is used for both unicast and multicast, the transport implementation will determine whether or not the same thread can be used to process both unicast and multicast data. For UDPv4, only one thread is needed per port–independent of whether the data was received via unicast or multicast data. See 21.3 Receive Threads on page 911 for more information.

### 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy

The TypeConsistency EnforcementQosPolicy defines the rules that determine whether the type used to publish a given topic is consistent with the type used to subscribe to it.
The QosPolicy structure includes the members in the following table.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TypeConsistencyKind</td>
<td>kind</td>
<td>Can be any of the following values:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AUTO_TYPE_COERCION (default)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ALLOW_TYPE_COERCION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DISALLOW_TYPE_COERCION</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See below for details.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>ignore_sequence_bounds</td>
<td>Controls whether sequence bounds are taken into consideration for type assignability. If false, a sequence with a larger maximum length may not be assigned to a sequence with a smaller maximum length. If true, sequences and strings in a DataReader type can have a maximum length smaller than that of the DataWriter type. When the length of the sequence in a particular sample is larger than the maximum length, that sample is discarded. Default: true</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>ignore_string_bounds</td>
<td>Controls whether string bounds are taken into consideration for type assignability. If false, then a string with a larger maximum length may not be assigned to a string with a smaller maximum length. Default: true</td>
</tr>
</tbody>
</table>
| DDS_Boolean                 | ignore_member_names   | Controls whether member names are taken into consideration for type assignability. If false, members with the same ID and different names are not assignable to each other. If true, members of a type can change their name while keeping their member ID. For example, MyType and MyTypeSpanish are only assignable if ignore_member_names is true:  

```c
struct MyType {
    @id(10) int32 x;
    @id(20) int32 angle;
};
struct MyTypeSpanish {
    @id(10) int32 x;
    @id(20) int32 angulo;
};
```

Default: false |
| DDS_Boolean                 | prevent_type_widening | Controls whether type widening is allowed. A type T2 widens a type T1 when T2 contains required members that are not present in T1. If a DataReader of T2 sets prevent_type_widening to true, then the DataReader will not be matched with a DataWriter of T1 because T1 is not assignable to T2. Default: false |
| DDS_Boolean                 | force_type_validation | Controls whether type information must be available in order to complete matching between a DataWriter and this DataReader. If false, matching may occur as long as the type names match. Note that if the types have the same name but are not assignable, DataReaders may fail to deserialize incoming data samples. Default: false |
Table 8.25 DDS_TypeConsistencyEnforcementQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>ignore_enum_literal_names</td>
<td>Controls whether enumeration constant names are taken into consideration for type assignability. If the option is set to true, then enumeration constants may change their names, but not their values, and still maintain assignability. If the option is set to false, then in order for enumerations to be assignable, any constant that has the same value in both enumerations must also have the same name. For example, enum Color {RED = 0} and enum Color {ROJO = 0} are assignable if and only if ignore_enum_literal_names is true. Default: false</td>
</tr>
</tbody>
</table>

The type-consistency enforcement rules consist of two steps:

1. If both the DataWriter and DataReader specify a TypeObject, it is considered first. If the DataReader allows type coercion, then its type must be assignable from the DataWriter’s type, taking into account the values of **prevent_type_widening, ignore_sequence_bounds, ignore_string_bounds, ignore_member_names, and ignore_enum_literal_names**. If the DataReader does not allow type coercion, then its type must be equivalent to the type of the DataWriter.

2. If either the DataWriter or the DataReader does not provide a TypeObject definition, then the registered type names are examined. The DataReader’s and DataWriter’s registered type names must match exactly, as was true in Connext DDS releases prior to 5.0. This step will fail if **force_type_validation** is true, regardless of the type names.

If either Step 1 or Step 2 fails, the Topics associated with the DataReader and DataWriter are considered to be inconsistent and the 5.3.1 INCONSISTENT_TOPIC Status on page 226 is updated.

The default enforcement kind is DDS_AUTO_TYPE_COERCION. This default kind translates to DDS_ALLOW_TYPE_COERCION, except in the following cases:

- When a Zero Copy DataReader is used, the kind is translated to DDS_DISALLOW_TYPE_COERCION.
- When the middleware is introspecting the built-in topic data declaration of a remote DataReader in order to determine whether it can match with a local DataWriter, if it observes that no TypeConsistency Enforcement QosPolicy value is provided (as would be the case when communicating with a Service implementation not in conformance with this specification), it assumes a kind of DDS_DISALLOW_TYPE_COERCION.

### 8.6.6.1 Values for TypeConsistencyKind

- **AUTO_TYPE_COERCION** (default)

  For a regular DataReader, this default value is translated to ALLOW_TYPE_COERCION. For a Zero Copy DataReader, this default value is translated to DISALLOW_TYPE_COERCION. (See
23.1.5 Zero Copy Transfer Over Shared Memory on page 962 for information on why a Zero Copy DataReader requires the DISALLOW_TYPE_COERCION option.)

- DISALLOW_TYPE_COERCION

With this setting, the DataWriter and DataReader must support the same data type in order for them to communicate. (This is the degree of enforcement required by the OMG DDS Specification prior to the OMG ‘Extensible and Dynamic Topic Types for DDS’ specification.)

When Connext DDS is introspecting the built-in topic data declaration of a remote DataWriter or DataReader, if no TypeConsistencyEnforcementQosPolicy value is provided (as would be the case when communicating with an implementation not in conformance with the Extensible and Dynamic Topic Types for DDS" (DDS-XTypes) specification), Connext DDS shall assume a kind of DISALLOW_TYPE_COERCION.

- ALLOW_TYPE_COERCION

With this setting, the DataWriter and the DataReader need not support the same data type in order for them to communicate, as long as the DataReader’s type is assignable from the DataWriter’s type.

For example, the following two extensible types will be assignable to each other since MyDerivedType contains all the members of MyBaseType (member_1) plus an additional element (member_2).

```
struct MyBaseType {
    int32 member_1;
};
struct MyDerivedType: MyBaseType {
    int32 member_2;
};
```

Even if MyDerivedType was not explicitly inherited from MyBaseType, the types would still be assignable. For example:

```
struct MyBaseType {
    int32 member_1;
};
struct MyDerivedType {
    int32 member_1;
    int32 member_2;
};
```

For more information, see the RTI Connext DDS Core Libraries Extensible Types Guide and the OMG ‘Extensible and Dynamic Topic Types for DDS’ Specification.
### 8.6.6.2 Prevent Type Widening

The `prevent_type_widening` field determines whether type widening is allowed. In Figure 8.25: `prevent_type_widening = false` below, `VehicleData_v2` has three members and `VehicleData_v1` two members. With type widening allowed, the narrower car (`VehicleData_v1`, with two members) can write to the wider car (`VehicleData_v2`), but notice that the `DataReader` assumes a value that might be misleading (in this case, a default speed of zero).

**Figure 8.25: prevent_type_widening = false**

```java
@appendable
struct Position {
    double latitude;
    double longitude;
};

@appendable
struct VehicleData_v2 {
    long vehicleID;
    Position position;
    double speed;
}

@appendable
struct VehicleData_v1 {
    long vehicleID;
    Position position;
}
```

If widening is not allowed (**Figure 8.26: prevent_type_widening = true on the next page**), `VehicleData_v1` and `VehicleData_v2` do not communicate with each other.
8.6.6.3 Properties

This QoS Policy cannot be modified after the *DataReader* is enabled.

It only applies to *DataReaders*, so there is no requirement that the publishing and subscribing sides use compatible values.

8.6.6.4 Related QoS Policies

- None.

8.6.6.5 Applicable Entities

- 8.3 *DataReaders* on page 504

8.6.6.6 System Resource Considerations

- None.
Chapter 9 Working with DDS Domains

This section discusses how to use DomainParticipants. It describes the types of operations that are available for them and their QosPolicies.

The goal of this section is to help you become familiar with the objects you need for setting up your Connext DDS application. For specific details on any mentioned operations, see the API Reference HTML documentation.

9.1 Fundamentals of DDS Domains and DomainParticipants

DomainParticipants are the focal point for creating, destroying, and managing other Connext DDS objects. A DDS domain is a logical network of applications: only applications that belong to the same DDS domain may communicate using Connext DDS. A DDS domain is identified by a unique integer value known as a domain ID. An application participates in a DDS domain by creating a DomainParticipant for that domain ID.
Applications can belong to multiple DDS domains—A belongs to DDS domains 1 and 2. Applications in the same DDS domain can communicate with each other, such as A and B, or A and C. Applications in different DDS domains, such as B and C, are not even aware of each other and will not exchange messages.

As seen in Figure 9.1: Relationship between Applications and DDS Domains above, a single application can participate in multiple DDS domains by creating multiple DomainParticipants with different domain IDs. DomainParticipants in the same DDS domain form a logical network; they are isolated from DomainParticipants of other DDS domains, even those running on the same set of physical computers sharing the same physical network. DomainParticipants in different DDS domains will never exchange messages with each other. Thus, a DDS domain establishes a “virtual network” linking all DomainParticipants that share the same domain ID.

An application that wants to participate in a certain DDS domain will need to create a DomainParticipant. As seen in Figure 9.2: DDS Domain Module on the next page, a DomainParticipant object is a container for all other Entities that belong to the same DDS domain. It acts as factory for the Publisher, Subscriber, and Topic entities. (As seen in Sending Data (Chapter 7 on page 270) and Receiving Data (Chapter 8 on page 484), in turn, Publishers are factories for DataWriters and Subscribers are factories for DataReaders.) DomainParticipants cannot contain other DomainParticipants.

Like all Entities, DomainParticipants have QosPolicies and Listeners. The DomainParticipant entity also allows you to set ‘default’ values for the QosPolicies for all the entities created from it or from the entities that it creates (Publishers, Subscribers, Topics, DataWriters, and DataReaders).
9.2 DomainParticipantFactory

- C, Traditional C++, Java and .NET APIs:

  The main purpose of a DomainParticipantFactory is to create and destroy DomainParticipants.

In C++ terms, this is a singleton class; that is, you will only have a single DomainParticipantFactory in an application—no matter how many DomainParticipants the application may create. Figure 9.3: Instantiating a DomainParticipantFactory on the next page shows how to
instantiate a DomainParticipantFactory. Notice that there are no parameters to specify. Alternatively, in C++, C++/CLI, and C#, the predefined macro, DDSTheParticipantFactory, can also be used to retrieve the singleton factory. (In C, the macro is DDS_TheParticipantFactory. In Java, use the static class method DomainParticipantFactory.TheParticipantFactory.)

Unlike the other Entities that you create, the DomainParticipantFactory does not have an associated Listener. However, it does have associated QosPolicies, see 9.2.1 Setting DomainParticipantFactory QosPolicies on page 605. You can change them using the factory’s get_qos() and set_qos() operations. The DomainParticipantFactory also stores the default QoS settings that can be used when a DomainParticipant is created. These default settings can be changed as well, see 9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627.

Figure 9.3: Instantiating a DomainParticipantFactory

```c++
DDSDomainParticipantFactory* factory = NULL;
factory = DDSDomainParticipantFactory::get_instance();
if (factory == NULL) {
    // ... error
}
```

- Modern C++ API:

  In the Modern C++ API, there isn’t a explicit DomainParticipantFactory. DomainParticipants are created using their constructors and are automatically destroyed as a reference type (See 4.1.1 Creating and Deleting DDS Entities on page 167).

  The operations to set and get the default DomainParticipantQos are static functions in DomainParticipant: DomainParticipant::default_participant_qos(). The operations to look up participants are freestanding functions in the dds::domain and rti::domain namespaces: dds::domain::find(), rti::domain::find_participant_by_name(), and rti::domain::find_participants(). The class QosProvider is responsible for managing QoS profiles (see 19.5 How to Load XML-Specified QoS Settings on page 882).

  There is a DomainParticipantFactoryQos, but it only contains the ENTITY_FACTORY to indicate if a DomainParticipant should be enabled in its constructor or by calling enable(), and SYSTEM_RESOURCE_LIMITS. The DomainParticipantFactoryQos getter and setter are static functions in DomainParticipant: DomainParticipant::participant_factory_qos().

  Another static function in DomainParticipant allows finalizing the implicit DomainParticipantFactory singleton: DomainParticipant::finalize_participant_factory().

Once you have a DomainParticipantFactory, you can use it to perform the operations listed in Table 9.1 DomainParticipantFactory Operations. The most important one is create_participant(), described in 9.3.1 Creating a DomainParticipant on page 615. For more details on all operations, see the API Reference HTML documentation as well as the section of the manual listed in the Reference column.
### Table 9.1 DomainParticipantFactory Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-Participants</td>
<td>create_participant</td>
<td>Creates a <em>DomainParticipant</em>.</td>
<td>9.3.1 Creating a DomainParticipant on page 615</td>
</tr>
<tr>
<td></td>
<td>create_participant_with_profile</td>
<td>Creates a <em>DomainParticipant</em> based on a QoS profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete_participant</td>
<td>Deletes a <em>DomainParticipant</em>.</td>
<td>9.3.2 Deleting DomainParticipants on page 617</td>
</tr>
<tr>
<td></td>
<td>get_default_participant_qos</td>
<td>Gets the default QoS for <em>DomainParticipants</em>.</td>
<td>9.2.2 Getting and Setting Default QoS for DomainParticipants on page 607</td>
</tr>
<tr>
<td></td>
<td>get_participants</td>
<td>Returns a sequence of pointers to all the <em>DomainParticipants</em> within the DomainParticipantFactory.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lookup_participant</td>
<td>Finds a specific <em>DomainParticipant</em>, based on a domain ID.</td>
<td>9.2.4 Looking Up DomainParticipants on page 608</td>
</tr>
<tr>
<td></td>
<td>lookup_participant_by_name</td>
<td>Finds a specific <em>DomainParticipant</em>, based on a domain name.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_participant_qos</td>
<td>Sets the default QoS for <em>DomainParticipants</em>.</td>
<td>9.2.2 Getting and Setting Default QoS for DomainParticipants on page 607</td>
</tr>
<tr>
<td></td>
<td>set_default_participant_qos_with_profile</td>
<td>Sets the default QoS for <em>DomainParticipants</em> based on a QoS profile.</td>
<td></td>
</tr>
<tr>
<td>The Factory’s Instance</td>
<td>get_instance</td>
<td>Gets the singleton instance of this class.</td>
<td>9.2.3 Freeing Resources Used by the DomainParticipantFactory on page 607</td>
</tr>
<tr>
<td></td>
<td>finalize_instance</td>
<td>Destroys the singleton instance of this class.</td>
<td></td>
</tr>
<tr>
<td>The Factory’s Own QoS</td>
<td>get_qos</td>
<td>Gets/sets the DomainParticipantFactory’s QoS.</td>
<td>4.1.7 Getting, Setting, and Comparing QosPolicies on page 172</td>
</tr>
<tr>
<td></td>
<td>set_qos</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Compares two DomainParticipantFactory’s QoS structures for equality.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 9.1 DomainParticipantFactory Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>set_thread_ factory</td>
<td>Specifies a ThreadFactory implementation that DomainParticipants will use to create and delete all threads.</td>
<td>21.7 User-Managed Threads on page 915</td>
</tr>
<tr>
<td></td>
<td>unregister_thread</td>
<td>Frees all resources related to a thread. This function is intended to be used at the end of any user-created threads that invoke Connext DDS APIs (not all users will have this situation). The best approach is to call it immediately before exiting such a thread, after all Connext DDS APIs have been called.</td>
<td></td>
</tr>
<tr>
<td>Profiles &amp; Libraries</td>
<td>get_default_library</td>
<td>Gets the default library for a DomainParticipantFactory.</td>
<td>9.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library on the next page</td>
</tr>
<tr>
<td></td>
<td>get_default_profile</td>
<td>Gets the default QoS profile for a DomainParticipantFactory.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_profile_library</td>
<td>Gets the library that contains the default QoS profile for a DomainParticipantFactory.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_&lt;entity&gt;_qos_from_profile</td>
<td>Gets the <code>&lt;entity&gt;</code> QoS values associated with a specified QoS profile, <code>&lt;entity&gt;</code> may be topic, datareader, datawriter, subscriber, publisher, or participant.</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_&lt;entity&gt;_qos_from_profile_w_topic_name</td>
<td>Like get_&lt;entity&gt;_qos_from_profile(), but this operation allows you to specify a topic name associated with the entity. The topic filter expressions in the profile will be evaluated on the topic name. <code>&lt;entity&gt;</code> may be topic, datareader, or datawriter.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_qos_profiles</td>
<td>Gets the names of all XML QoS profiles associated with a specified XML QoS profile library.</td>
<td>19.4 Tags for Configuring QoS with XML on page 874</td>
</tr>
<tr>
<td></td>
<td>get_qos_profile_library</td>
<td>Gets the names of all XML QoS profile libraries associated with the DomainParticipantFactory.</td>
<td>19.10.1 Retrieving a List of Available Libraries on page 894</td>
</tr>
<tr>
<td></td>
<td>load_profiles</td>
<td>Explicitly loads or reloads the QoS profiles.</td>
<td>19.5.1 Loading, Reloading and Unloading Profiles on page 884</td>
</tr>
<tr>
<td></td>
<td>reload_profiles</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_profile</td>
<td>Sets the default QoS profile for a DomainParticipantFactory.</td>
<td>9.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library on the next page</td>
</tr>
<tr>
<td></td>
<td>set_default_library</td>
<td>Sets the default library for a DomainParticipantFactory.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>unload_profiles</td>
<td>Frees the resources associated with loading QoS profiles.</td>
<td>19.5.1 Loading, Reloading and Unloading Profiles on page 884</td>
</tr>
</tbody>
</table>

### 9.2.1 Setting DomainParticipantFactory QosPolicies

The DDS_DomainParticipantFactoryQos structure has the following format:
For information on why you would want to change a particular QoSProfile, see the section referenced in 
Table 9.2 DomainParticipantFactory QoS.

### Table 9.2 DomainParticipantFactory QoS

<table>
<thead>
<tr>
<th>QoSPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntityFactory</td>
<td>Controls whether or not child entities are created in the enabled state. See 7.4.2 ENTITYFACTORY QoSPolicy on page 348.</td>
</tr>
<tr>
<td>Logging</td>
<td>Configures the properties associated with Connext DDS logging. See 9.4.1 LOGGING QoSPolicy (DDS Extension) on page 632.</td>
</tr>
<tr>
<td>Profile</td>
<td>Configures the way that XML documents containing QoS profiles are loaded by RTI. See 9.4.2 PROFILE QoSPolicy (DDS Extension) on page 633.</td>
</tr>
<tr>
<td>SystemResource-Limits</td>
<td>Configures DomainParticipant-independent resources used by Connext DDS. Mainly used to change the maximum number of DomainParticipants that can be created within a single process (address space). See 9.4.3 SYSTEM_RESOURCE_LIMI TES QoS Policy (DDS Extension) on page 635.</td>
</tr>
</tbody>
</table>

### 9.2.1.1 Getting and Setting the DomainParticipantFactory’s Default QoS Profile and Library

You can retrieve the default QoS profile for the DomainParticipantFactory with the `get_default_profile()` operation. You can also get the default library for the DomainParticipantFactory, as well as the library that contains the DomainParticipantFactory’s default profile (these are not necessarily the same library); these operations are called `get_default_library()` and `get_default_library_profile()`, respectively. These operations are for informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see Configuring QoS with XML (Chapter 19 on page 848).

```c
virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_library_profile ()
```

There are also operations for setting the DomainParticipantFactory’s default library and profile:

```c
DDS_ReturnCode_t set_default_library (const char * library_name)
DDS_ReturnCode_t set_default_profile (const char * library_name,
const char * profile_name)
```

`set_default_profile()` specifies the profile that will be used as the default the next time a default DomainParticipantFactory profile is needed during a call to a DomainParticipantFactory operation.

When calling a DomainParticipantFactory operation that requires a `profile_name` parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)
The `set_default_profile()` operation does not set the default QoS for the `DomainParticipant` that can be created by the `DomainParticipantFactory`. To set the default QoS using a profile, use the `DomainParticipantFactory`’s `set_default_participant_qos_with_profile()` operation (see 9.2.2 Getting and Setting Default QoS for DomainParticipants below).

### 9.2.2 Getting and Setting Default QoS for DomainParticipants

To get the default QoS that will be used for creating `DomainParticipants` if `create_participant()` is called with `DDS_PARTICIPANT_QOS_DEFAULT` as the `qos` parameter, use this `DomainParticipantFactory` operation:

```
DDS_ReturnCode_t get_default_participant_qos (DDS_DomainParticipantQos &qos)
```

This operation gets the QoS settings that were specified on the last successful call to `set_default_participant_qos()` or `set_default_participant_qos_with_profile()`, or if the call was never made, the default values listed in DDS_DomainParticipantQos.

To set the default QoS that will be used for new `DomainParticipants`, use the following operations. Then these default QoS will be used if `create_participant()` is called with `DDS_PARTICIPANT_QOS_DEFAULT` as the ‘qos’ parameter.

```
DDS_ReturnCode_t set_default_participant_qos (const DDS_DomainParticipantQos &qos)
```

or

```
DDS_ReturnCode_t set_default_participant_qos_with_profile (const char *library_name, const char *profile_name)
```

**Notes:**

- These operations may potentially allocate memory, depending on the sequences contained in some QoS policies.
- It is not safe to set the default `DomainParticipant` QoS values while another thread may be simultaneously calling `get_default_participant_qos()`, `set_default_participant_qos()`, or `create_participant()` with `DDS_PARTICIPANT_QOS_DEFAULT` as the `qos` parameter. It is also not safe to get the default `DomainParticipant` QoS values while another thread may be simultaneously calling `set_default_participant_qos()`.

### 9.2.3 Freeing Resources Used by the DomainParticipantFactory

The `finalize_instance()` operation explicitly reclaims resources used by the participant factory singleton (including resources use for QoS profiles).

On many operating systems, these resources are automatically reclaimed by the OS when the program terminates. However, some memory-check tools will flag those resources as unreclaimed. This method provides a way to clean up all the memory used by the participant factory.
Before calling finalize_instance() on a DomainParticipantFactory, all of the participants created by the factory must have been deleted. For a DomainParticipant to be successfully deleted, all Entities created by the participant or by the Entities that the participant created must have been deleted. In essence, the DomainParticipantFactory cannot be deleted until all other Entities have been deleted in an application.

Except for Linux systems: get_instance() and finalize_instance() are UNSAFE on the FIRST call. It is not safe for two threads to simultaneously make the first call to get or finalize the factory instance. Subsequent calls are thread safe.

### 9.2.4 Looking Up DomainParticipants

The DomainParticipantFactory has these useful operations for retrieving its DomainParticipants:

- **get_participants()** returns a sequence of pointers to all the DomainParticipants within the DomainParticipantFactory.

```
DDS_ReturnCode_t
    get_participants (DDSDomainParticipantSeq & participants)
```

- **lookup_participant()** locates an existing DomainParticipant based on its domain ID.

```
DDSDomainParticipant *
    lookup_participant (DDS_DomainId_t domainId)
```

- **lookup_participant_by_name()** locates an existing DomainParticipant based on its name.

```
DDSDomainParticipant *
    lookup_participant_by_name(const char * participant_name)
```

Note: in the Modern C++ API these operations are freestanding functions rti::domain::find_participants(), dds::domain::find(), and rti::domain::find_participant_by_name(), respectively.

### 9.2.5 Getting QoS Values from a QoS Profile

A QoS Profile may include configuration settings for all types of Entities. If you just want the settings for a specific type of Entity, call get_<entity>qos_from_profile() (where <entity> may be participant, publisher, subscriber, datawriter, datareader, or topic). This is useful if you want to get the QoS values from the profile in a structure, make some changes, and then use that structure to create an entity.

```
DDS_ReturnCode_t get_<entity>qos_from_profile (    
    DDS_<Entity>Qos &qos,    
    const char *library_name,    
    const char *profile_name)
```

For an example, see Figure 7.5: Getting QoS Values from a Profile, Changing QoS Values, Creating a Publisher with Modified QoS Values on page 280.
The `get_<entity>_qos_from_profile()` operations do not take into account the `topic_filter` attributes that may be set for `DataWriter`, `DataReader`, or `Topic` QoSs in profiles (see 19.3.4 Topic Filters on page 870). If there is a topic name associated with an entity, you can call `get_<entity>_qos_from_profile_w_topic_name()` (where `<entity>` can be datawriter, datareader, or topic) and the topic filter expressions in the profile will be evaluated on the topic name.

```c
DDS_ReturnCode_t get_<entity>_qos_from_profile_w_topic_name(
    DDS_<entity>Qos &qos,
    const char *library_name,
    const char *profile_name,
    const char *topic_name)
```

`get_<entity>_qos_from_profile()` and `get_<entity>_qos_from_profile_w_topic_name()` may allocate memory, depending on the sequences contained in some QoS policies.

Note: in the Modern C++ API, the class QosProvider provides the functionality described in this section. Please see the API Reference HTML documentation: Modules, RTI Connext DDS API Reference, Configuring QoS Profiles with XML, QosProvider.

### 9.3 DomainParticipants

A `DomainParticipant` is a container for `Entity` objects that all belong to the same DDS domain. Each `DomainParticipant` has its own set of internal threads and internal data structures that maintain information about the `Entities` created by itself and other `DomainParticipants` in the same DDS domain. A `DomainParticipant` is used to create and destroy `Publishers`, `Subscribers` and `Topics`.

Once you have a `DomainParticipant`, you can use it to perform the operations listed in Table 9.3 DomainParticipant Operations. For more details on all operations, see the API Reference HTML documentation. Some of the first operations you’ll be interested in are `create_topic()`, `create_subscriber()`, and `create_publisher()`.

**Note:** Some operations cannot be used within a listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Builtin Subscriber</td>
<td>get_builtin_subscriber</td>
<td>Returns the builtin Subscriber.</td>
<td>18.2 Built-in DataReaders on page 832</td>
</tr>
<tr>
<td>Working with ...</td>
<td>Operation</td>
<td>Description</td>
<td>Reference</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DomainParticipants</td>
<td>add_peer</td>
<td>Adds an entry to the peer list.</td>
<td>9.5.2.3 Adding and Removing Peers List Entries on page 641</td>
</tr>
<tr>
<td></td>
<td>enable</td>
<td>Enables the DomainParticipant.</td>
<td>4.1.2 Enabling DDS Entities on page 168</td>
</tr>
<tr>
<td></td>
<td>equals</td>
<td>Compares two DomainParticipant’s QoS structures for equality.</td>
<td>9.3.7.2 Comparing QoS Values on page 625</td>
</tr>
<tr>
<td></td>
<td>get_discovered_participant_data</td>
<td>Provides the ParticipantBuiltInTopicData for a discovered DomainParticipant</td>
<td>9.3.12 Learning about Discovered DomainParticipants on page 630</td>
</tr>
<tr>
<td></td>
<td>get_discovered_participants</td>
<td>Provides a list of DomainParticipants that have been discovered.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_domain_id</td>
<td>Gets the domain ID of the DomainParticipant.</td>
<td>9.3.4 Choosing a Domain ID and Creating Multiple DDS Domains on page 618</td>
</tr>
<tr>
<td></td>
<td>get_listener</td>
<td>Gets the currently installed DomainParticipantListener.</td>
<td>9.3.6 Setting Up DomainParticipantListeners on page 620</td>
</tr>
<tr>
<td></td>
<td>get_qos</td>
<td>Gets the DomainParticipant QoS.</td>
<td>9.3.7 Setting DomainParticipantQosPolicies on page 622</td>
</tr>
<tr>
<td></td>
<td>ignore_participant</td>
<td>Rejects the connection to a remote DomainParticipant.</td>
<td>18.4 Restricting Communication—Ignoring Entities on page 841</td>
</tr>
<tr>
<td></td>
<td>remove_peer</td>
<td>Removes an entry from the peer list.</td>
<td>9.5.2.3 Adding and Removing Peers List Entries on page 641</td>
</tr>
<tr>
<td></td>
<td>set_listener</td>
<td>Replaces the DomainParticipantListener.</td>
<td>9.3.6 Setting Up DomainParticipantListeners on page 620</td>
</tr>
<tr>
<td></td>
<td>set_qos</td>
<td>Sets the DomainParticipant QoS.</td>
<td>9.3.7 Setting DomainParticipantQosPolicies on page 622</td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>Sets the DomainParticipant QoS based on a QoS profile.</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.3 DomainParticipant Operations
### Table 9.3 DomainParticipant Operations

<table>
<thead>
<tr>
<th>Working with</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content-Filtered-Topics</td>
<td>create_con-tenfilteredtopic</td>
<td>Creates a ContentFilteredTopic that can be used to process content-based subscriptions.</td>
<td>5.4.3 Creating ContentFilteredTopics on page 229</td>
</tr>
<tr>
<td></td>
<td>create_con-tenfilteredtopic_ with_filter</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete_con-tenfilteredtopic</td>
<td>Deletes a ContentFilteredTopic.</td>
<td>5.4.4 Deleting ContentFilteredTopics on page 233</td>
</tr>
<tr>
<td></td>
<td>register_contentfilter</td>
<td>Registers a new content filter.</td>
<td>5.4.10.2 Registering a Custom Filter on page 249</td>
</tr>
<tr>
<td></td>
<td>unregister_contentfilter</td>
<td>Unregisters a new content filter.</td>
<td>5.4.10.3 Unregistering a Custom Filter on page 251</td>
</tr>
<tr>
<td></td>
<td>lookup_contentfilter</td>
<td>Gets a previously registered content filter.</td>
<td>5.4.10.4 Retrieving a ContentFilter on page 252</td>
</tr>
<tr>
<td>DataReaders</td>
<td>create_datareader</td>
<td>Creates a DataReader with a given DataReaderListener, and an implicit Subscriber.</td>
<td>8.3.1 Creating DataReaders on page 510</td>
</tr>
<tr>
<td></td>
<td>create_datareader_ with_ profile</td>
<td>Creates a DataReader based on a QoS profile, with a given DataReaderListener, and an implicit Subscriber.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete_datareader</td>
<td>Deletes a DataReader that belongs to the implicit Subscriber.</td>
<td>8.3.3 Deleting DataReaders on page 512</td>
</tr>
<tr>
<td></td>
<td>get_default_datareader_qos</td>
<td>Copies the default DataReaderQoS values into the provided structure.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ignore_subscription</td>
<td>Rejects the connection to a DataReader</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_datareader_qos</td>
<td>Sets the default DataReaderQos values.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_datareader_qos_with_profile</td>
<td>Sets the default DataReaderQos using values from a QoS profile.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 9.3 DomainParticipant Operations

<table>
<thead>
<tr>
<th>Working with</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DataWriters</strong></td>
<td>create_datawriter</td>
<td>Creates a DataWriter with a given DataWriterListener, and an implicit Publisher.</td>
<td>7.2.2 Creating Publishers on page 276</td>
</tr>
<tr>
<td></td>
<td>create_datawriter_with_profile</td>
<td>Creates a DataWriter based on a QoS profile, with a given DataWriterListener, and an implicit Publisher.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete_datawriter</td>
<td>Deletes a DataWriter that belongs to the ‘implicit Publisher.’</td>
<td>7.2.3 Deleting Publishers on page 277</td>
</tr>
<tr>
<td></td>
<td>ignore_publication</td>
<td>Rejects the connection to a DataWriter.</td>
<td>18.4 Restricting Communication—Ignoring Entities on page 841</td>
</tr>
<tr>
<td></td>
<td>get_default_datawriter_qos</td>
<td>Copies the default DataWriterQos values into the provided DataWriterQos structure.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_datawriter_qos</td>
<td>Sets the default DataWriterQos values.</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_datawriter_qos_with_profile</td>
<td>Sets the default DataWriterQos using values from a profile.</td>
<td></td>
</tr>
<tr>
<td><strong>Publishers</strong></td>
<td>create_publisher</td>
<td>Creates a Publisher and a PublisherListener.</td>
<td>7.2.2 Creating Publishers on page 276</td>
</tr>
<tr>
<td></td>
<td>create_publisher_with_profile</td>
<td>Creates a Publisher based on a QoS profile, and a PublisherListener.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete_publisher</td>
<td>Deletes a Publisher.</td>
<td>7.2.3 Deleting Publishers on page 277</td>
</tr>
<tr>
<td></td>
<td>get_default_publisher_qos</td>
<td>Copies the default PublisherQos values into the provided PublisherQos structure.</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>get_implicit_publisher</td>
<td>Gets the Publisher that is implicitly created by the DomainParticipant.</td>
<td>9.3.10 Getting the Implicit Publisher or Subscriber on page 629</td>
</tr>
<tr>
<td></td>
<td>get_publishers</td>
<td>Provides a list of all Publishers owned by the DomainParticipant.</td>
<td>9.3.15.3 Getting All Publishers and Subscribers on page 632</td>
</tr>
<tr>
<td></td>
<td>set_default_publisher_qos</td>
<td>Sets the default PublisherQos values.</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_publisher_qos_with_profile</td>
<td>Sets the default PublisherQos using values from a QoS profile.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 9.3 DomainParticipant Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>create_subscriber</td>
<td>Creates a Subscriber and a SubscriberListener.</td>
<td>8.2.2 Creating Subscribers on page 491</td>
</tr>
<tr>
<td></td>
<td>create_subscriber_with_profile</td>
<td>Creates a Subscriber based on a QoS profile, and a SubscriberListener.</td>
<td>8.2.3 Deleting Subscribers on page 492</td>
</tr>
<tr>
<td>Subscribers</td>
<td>delete_subscriber</td>
<td>Deletes a Subscriber.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_subscriber_qos</td>
<td>Copies the default SubscriberQos values into the provided SubscriberQos structure.</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>get_implicit_subscriber</td>
<td>Gets the Subscriber that is implicitly created by the DomainParticipant.</td>
<td>9.3.10 Getting the Implicit Publisher or Subscriber on page 629</td>
</tr>
<tr>
<td></td>
<td>get_subscribers</td>
<td>Provides a list of all Subscribers owned by the DomainParticipant.</td>
<td>9.3.15.3 Getting All Publishers and Subscribers on page 632</td>
</tr>
<tr>
<td></td>
<td>set_default_subscriber_qos</td>
<td>Sets the default SubscriberQos values.</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_subscriber_qos_with_profile</td>
<td>Sets the default SubscriberQos values using values from a QoS profile.</td>
<td></td>
</tr>
<tr>
<td>Durable Subscriptions</td>
<td>delete_durable_subscription</td>
<td>Deletes an existing Durable Subscription. The quorum of the existing DDS samples will be considered satisfied.</td>
<td>46.9 Configuring Durable Subscriptions in Persistence Service on page 1200</td>
</tr>
</tbody>
</table>
|                  | register_durable_subscription                | Creates a Durable Subscription that will receive all DDS samples published on a Topic, including those published while a DataReader is inactive or before it may be created.  
RTI Persistence Service will ensure that all the DDS samples on that Topic are retained until they are acknowledged by at least N DataReaders belonging to the Durable Subscription, where N is the quorum count.  
If the same Durable Subscription is created on a different Topic, RTI Persistence Service will implicitly delete the previous Durable Subscription and create a new one on the new Topic. |                                                                                                                                                         |
## Table 9.3 DomainParticipant Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Topics</strong></td>
<td>create_topic</td>
<td>Creates a Topic and a TopicListener.</td>
<td>5.1.1 Creating Topics on page 217</td>
</tr>
<tr>
<td></td>
<td>create_topic_with_profile</td>
<td>Creates a Topic based on a QoS profile, and a TopicListener.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>delete_topic</td>
<td>Deletes a Topic.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_topic_qos</td>
<td>Copies the default TopicQos values into the provided TopicQos structure.</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>get_discovered_topic_data</td>
<td>Retrieves the BuiltinTopicData for a discovered Topic.</td>
<td>9.3.13 Learning about Discovered Topics on page 630</td>
</tr>
<tr>
<td></td>
<td>get_discovered_topics</td>
<td>Returns a list of all (non-ignored) discovered Topics.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ignore_topic</td>
<td>Rejects a remote topic.</td>
<td>18.4 Restricting Communication—Ignoring Entities on page 841</td>
</tr>
<tr>
<td></td>
<td>lookup_topic_description</td>
<td>Gets an existing locally-created TopicDescription (Topic).</td>
<td>9.3.8 Looking up Topic Descriptions on page 628</td>
</tr>
<tr>
<td></td>
<td>set_default_topic_qos</td>
<td>Sets the default TopicQos values.</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_topic_qos_with_profile</td>
<td>Sets the default TopicQos values using values from a profile.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>find_topic</td>
<td>Finds an existing Topic, based on its name.</td>
<td>9.3.9 Finding a Topic on page 628</td>
</tr>
<tr>
<td><strong>Flow-Controllers</strong></td>
<td>create_flow-controller</td>
<td>Creates a custom FlowController object.</td>
<td>7.6.6 Creating and Deleting FlowControllers on page 481</td>
</tr>
<tr>
<td></td>
<td>delete_flow-controller</td>
<td>Deletes a custom FlowController object.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_flow-controller_property</td>
<td>Gets the default properties used when a new FlowController is created.</td>
<td>7.6.7 Getting/Setting Default FlowController Properties on page 482</td>
</tr>
<tr>
<td></td>
<td>set_default_flow-controller_property</td>
<td>Sets the default properties used when a new FlowController is created.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>lookup_flow-controller</td>
<td>Finds a FlowController, based on its name.</td>
<td>7.6.10 Other FlowController Operations on page 483</td>
</tr>
</tbody>
</table>
9.3.1 Creating a DomainParticipant

Typically, you will only need to create one DomainParticipant per DDS domain per application. (Although unusual, you can create multiple DomainParticipants for the same DDS domain in an application.)

To create a DomainParticipant, use the DomainParticipantFactory’s create_participant() or create_participant_with_profile() operation:

A QoS profile is way to use QoS settings from an XML file or string. With this approach, you can change QoS settings without recompiling the application. For details, see Configuring QoS with XML (Chapter 19 on page 848).

Note: In the Modern C++ API, you will use the DomainParticipant constructors.

```cpp
DDSDomainParticipant * create_participant(
  DDS_DomainId_t domainId,
  const DDS_DomainParticipantQos &qos,
```

### Table 9.3 DomainParticipant Operations

<table>
<thead>
<tr>
<th>Working with ...</th>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Libraries and Profiles</td>
<td>get_default_library</td>
<td>Gets the default library.</td>
<td>9.3.7.4 Getting and Setting DomainParticipant's Default QoS Profile and Library on page 626</td>
</tr>
<tr>
<td>Libraries and Profiles</td>
<td>get_default_profile</td>
<td>Gets the default profile.</td>
<td></td>
</tr>
<tr>
<td>Libraries and Profiles</td>
<td>get_default_profile_library</td>
<td>Gets the library that contains the default profile.</td>
<td></td>
</tr>
<tr>
<td>Libraries and Profiles</td>
<td>set_default_profile</td>
<td>Sets the default QoS profile.</td>
<td></td>
</tr>
<tr>
<td>Libraries and Profiles</td>
<td>set_default_library</td>
<td>Sets the default library.</td>
<td></td>
</tr>
<tr>
<td>MultiTopics</td>
<td>create_multitopic</td>
<td>Creates a MultiTopic that can be used to subscribe to multiple topics and combine/filter the received data into a resulting type.</td>
<td>Currently not supported.</td>
</tr>
<tr>
<td>MultiTopics</td>
<td>delete_multitopic</td>
<td>Deletes a MultiTopic.</td>
<td></td>
</tr>
<tr>
<td>Other</td>
<td>assert_liveliness</td>
<td>Manually asserts the liveliness of this DomainParticipant.</td>
<td>9.3.10 Getting the Implicit Publisher or Subscriber on page 629</td>
</tr>
<tr>
<td>Other</td>
<td>delete_contained_entities</td>
<td>Recursively deletes all the entities that were created using the &quot;create&quot; operations on the DomainParticipant and its children.</td>
<td>9.3.3 Deleting Contained Entities on page 617</td>
</tr>
<tr>
<td>Other</td>
<td>contains_entity</td>
<td>Confirms if an entity belongs to the DomainParticipant or not.</td>
<td>9.3.15.1 Verifying Entity Containment on page 631</td>
</tr>
<tr>
<td>Other</td>
<td>get_current_time</td>
<td>Gets the current time used by Connext DDS.</td>
<td>9.3.15.2 Getting the Current Time on page 631</td>
</tr>
<tr>
<td>Other</td>
<td>get_status_changes</td>
<td>Gets a list of statuses that have changed since the last time the application read the status or the Listeners were called.</td>
<td>4.1.4 Getting Status and Status Changes on page 171</td>
</tr>
</tbody>
</table>
Where:

- **domainId**: The domain ID uniquely identifies the DDS domain that the DomainParticipant is in. It controls with which other DomainParticipants it will communicate. See 9.3.4 Choosing a Domain ID and Creating Multiple DDS Domains on page 618 for more information on domain IDs.

- **qos**: If you want the default QoS settings (described in the API Reference HTML documentation), use DDS_PARTICIPANT_QOS_DEFAULT for this parameter (see Figure 9.4: Creating a DomainParticipant with Default QoS Policies below). If you want to customize any of the QoS Policies, supply a DomainParticipantQos structure that is described in 9.3.7 Setting DomainParticipant QoS Policies on page 622.

  **Note**: If you use DDS_PARTICIPANT_QOS_DEFAULT, it is not safe to create the DomainParticipant while another thread may simultaneously be calling the DomainParticipantFactory's set_default_participant_qos() operation.

- **listener**: Listeners are callback routines. Connext DDS uses them to notify your application of specific events (status changes) that may occur. The listener parameter may be set to NULL if you do not want to install a Listener. The DomainParticipant's Listener is a catchall for all of the events of all of its Entities. If an event is not handled by an Entity's Listener, then the DomainParticipantListener may be called in response to the event. For more information, see 9.3.6 Setting Up DomainParticipantListeners on page 620.

- **mask**: This bit mask indicates which status changes will cause the Listener to be invoked. The bits set in the mask must have corresponding callbacks implemented in the Listener. If you use NULL for the Listener, use DDS_STATUS_MASK_NONE for this parameter. If the Listener implements all callbacks, use DDS_STATUS_MASK_ALL. For information on statuses, see 4.4 Listeners on page 189.

- **library_name**: A QoS Library is a named set of QoS profiles. See 19.3 QoS Profiles on page 850.

- **profile_name**: A QoS profile groups a set of related QoS, usually one per entity. See 19.3 QoS Profiles on page 850.

  After you create a DomainParticipant, the next step is to register the data types that will be used by the application, see 3.6 Using RTI Code Generator (rtiddsgen) on page 152. Then you will need to create the Topics that the application will publish and/or subscribe, see 5.1.1 Creating Topics on page 217. Finally, you will use the DomainParticipant to create Publishers and/or Subscribers, see 7.2.2 Creating Publishers on page 276 and 8.2.2 Creating Subscribers on page 491.

  **Note**: It is not safe to create one DomainParticipant while another thread may simultaneously be looking up (9.2.4 Looking Up DomainParticipants on page 608) or deleting (9.3.2 Deleting DomainParticipants on the next page) the same DomainParticipant.

  For more examples, see 9.3.7.1 Configuring QoS Settings when DomainParticipant is Created on page 623.

#### Figure 9.4: Creating a DomainParticipant with Default QoS Policies

```c
DDSDomainParticipantListener *listener,
DSS_StatusMask mask)
DDSDomainParticipant * create_participant_with_profile(
    DDS_DomainId_t domainId,
    const char *library_name,
    const char *profile_name,
    DDSDomainParticipantListener *listener,
    DDS_StatusMask mask)
```
new MyDomainParticipantListener(); // or = NULL

// Create the participant
DDSDomainParticipant* participant = factory->create_participant(
  domain_id, DDS_PARTICIPANT_QOS_DEFAULT,
  participant_listener, DDS_STATUS_MASK_ALL);
if (participant == NULL) {
  // ... error
};

9.3.2 Deleting DomainParticipants

If the application is no longer interested in communicating in a certain DDS domain, the DomainParticipant can be deleted. A DomainParticipant can be deleted only after all the Entities that were created by the DomainParticipant have been deleted (see 9.3.3 Deleting Contained Entities below).

To delete a DomainParticipant:

You must first delete all Entities (Publishers, Subscribers, ContentFilteredTopics, and Topics) that were created with the DomainParticipant. Use the DomainParticipant’s delete_<entity>() operations to delete them one at a time, or use the delete_contained_entities() operation (9.3.3 Deleting Contained Entities below) to delete them all at the same time.

Delete the DomainParticipant by using the DomainParticipantFactory’s delete_participant() operation.

Note: A DomainParticipant cannot be deleted within its Listener callback, see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

After a DomainParticipant has been deleted, all of the participant’s internal Connext DDS threads and allocated memory will have been deleted. You should delete the DomainParticipantListener only after the DomainParticipant itself has been deleted.

Note: In the Modern C++ API, Entities are automatically destroyed.

9.3.3 Deleting Contained Entities

The DomainParticipant’s delete_contained_entities() operation deletes all the Publishers (including an implicitly created one, if it exists), Subscribers (including an implicitly created one, if it exists), ContentFilteredTopics, and Topics that have been created by the DomainParticipant.

Prior to deleting each contained entity, this operation recursively calls the corresponding delete_contained_entities() operation on each contained entity (if applicable). This pattern is applied recursively.
9.3.4 Choosing a Domain ID and Creating Multiple DDS Domains

Therefore, delete_contained_entities() on the DomainParticipant will end up deleting all the entities recursively contained in the DomainParticipant, that is also the DataWriter, DataReader, as well as the QueryCondition and ReadCondition objects belonging to the contained DataReader.

If delete_contained_entities() returns successfully, the application may delete the DomainParticipant knowing that it has no contained entities (see 9.3.2 Deleting DomainParticipants on the previous page).

9.3.4 Choosing a Domain ID and Creating Multiple DDS Domains

A domain ID identifies the DDS domain in which the DomainParticipant is communicating. DomainParticipants with the same domain ID are on the same communication “channel”. DomainParticipants with different domain IDs are completely isolated from each other.

The domain ID is a purely arbitrary value; you can use any integer 0 or higher, provided it does not violate the guidelines for the DDS_RtpsWellKnownPorts_t structure (9.5.9.2 Ports Used for Discovery on page 672). Domain IDs are typically between 0 and 232. Please see the API Reference HTML documentation for the DDS_RtpsWellKnownPorts_t structure and in particular, DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS.

Note: On Windows, you should avoid using ports 49152 through 65535 for inbound traffic. Connext DDS’s ephemeral ports (see 15.5 Ports Used for Discovery on page 791) may be within that range (see https://msdn.microsoft.com/en-us/library/windows/desktop/ms737550(v=vs.85).aspx). With the default RtpsWellKnownPorts settings, port 49152 corresponds to domain ID 167, so using domain IDs 168 through 232 on Windows introduces the risk of a port collision and failure to create the DomainParticipant when using multicast discovery. You may see this error:

```
RTI0sapiSocket_bindWithIP:OS bind() failure, error 0x271d: An attempt was made to access a socket in a way forbidden by its access permissions.
```

Most distributed systems can use a single DDS domain for all of its applications. Thus a single domain ID is sufficient. Some systems may need to logically partition nodes to prevent them from communicating with each other directly, and thus will need to use multiple DDS domains. However, even in systems that only use a single DDS domain, during the testing and development phases, one may want to assign different users/testers different domain IDs for running their applications so that their tests do not interfere with each other.

To run multiple applications on the same node with the same domain ID, Connext DDS uses a participant ID to distinguish between the different DomainParticipants in the different applications. The participant ID is simply an integer value that must be unique across all DomainParticipants created on the same node that use the same domain ID. The participant_id is part of the 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670.

Although usually those DomainParticipants have been created in different applications, the same application can also create multiple DomainParticipants with the same domain ID. For optimal results, the participant_id should be assigned sequentially to the different DomainParticipants, starting from the default value of 0.
Once you have a `DomainParticipant`, you can retrieve its domain ID with the `get_domain_id()` operation. The domain ID and participant ID are mapped to port numbers that are used by transports for discovery traffic. For information on how port numbers are calculated, see 15.5 Ports Used for Discovery on page 791. How `DomainParticipants` discover each other is discussed in Discovery (Chapter 15 on page 764).

### 9.3.5 Choosing a Domain Tag

The Domain Tag is an intuitive way of subdividing domains. It consists of a string value (with a maximum of 255 characters). It allows `DomainParticipants` to drop Participant Discovery messages not belonging to the same Domain Tag they are using. The Domain Tag is immutable, and cannot be changed after creating the `DomainParticipant`.

Unlike Domain IDs, Domain Tags are not mapped to port numbers that are used by transports for discovery traffic. Consequently, a `DomainParticipant` may receive Participant Discovery traffic belonging to a different Domain Tag; however, this traffic will be dropped upon reception. Another consequence of Domain Tag having no impact on port mapping is that multiple `DomainParticipants` running on the same machine with the same Domain ID, but with different Domain Tags, will end up using different participant IDs to avoid port collision.

As an example, a system with six `DomainParticipants` could be configured as follows:

- Participant A: Domain ID = 24, Domain Tag = "ENG. DEPT"
- Participant B: Domain ID = 24, Domain Tag = "ENG. DEPT"
- Participant C: Domain ID = 24, Domain Tag = "SALES DEPT"
- Participant D: Domain ID = 24, Domain Tag = "SALES DEPT"
- Participant E: Domain ID = 42, Domain Tag = "ENG. DEPT"
- Participant F: Domain ID = 42, Domain Tag = "ENG. DEPT"

In this system, Participants A, B, C, and D are all on the same Domain ID, so all of them will receive the discovery traffic belonging to domain 24; however, only `DomainParticipant` pairs A-B and C-D will be able to discover each other, since they have a matching Domain Tag. (Any discovery message not matching the expected Domain Tag will be dropped.) Participants E and F are in a different domain (42), so they are completely isolated from the rest, not even receiving the discovery traffic from the rest of the `DomainParticipants`. For more information, see 15.1.1 Simple Participant Discovery on page 765.

By default, a `DomainParticipant` is in an empty ("", zero-length string) Domain Tag. To associate a Domain Tag with a `DomainParticipant`, use the following `DomainParticipant` PropertyQos property:

```
dds.domain_participant.domain_tag: A string (with a maximum of 255 characters) defining the
Domain Tag the `DomainParticipant` will propagate through Participant Discovery. Participants will
drop any Participant discovery message that contains a Domain Tag that does not match the local
```

619
Domain Tag. This parameter is only propagated if it is set to a value different than the default. Default: "" (empty, zero-length string).

**Note:** While Domain ID is fully supported across the whole Connext DDS ecosystem, Domain Tag support is currently limited to the Core Libraries and infrastructure Services (by setting the aforementioned DomainParticipant PropertyQos property). Domain Tags are not well supported in Connext tools (such as Admin Console). Connext tools do not provide a tool-specific mechanism to configure Domain Tags. Consequently, if you configure an application to use Domain Tags, that application will not be able to communicate with Connext tools, unless you edit the tool's QoS configuration (if it has one—for instance, see Admin Console's Preferences dialog) to use Domain Tags.

### 9.3.6 Setting Up DomainParticipantListeners

DomainParticipants may optionally have Listeners. Listeners are essentially callback routines and are how Connext DDS will notify your application of specific events (changes in status) for entities Topics, Publishers, Subscribers, DataWriters, and DataReaders. Each Entity may have a Listener installed and enabled to process the events for itself and all of the sub-Entities created from it. If an Entity does not have a Listener installed or is not enabled to listen for a particular event, then Connext DDS will propagate the event to the Entity’s parent. If the parent Entity does not process the event, Connext DDS will continue to propagate the event up the object hierarchy until either a Listener is invoked or the event is dropped.

The DomainParticipantListener is the last chance that an event can be processed for the Entities descended from a DomainParticipant. The DomainParticipantListener is used only if an event is not handled by any of the Entities contained by the participant.

A Listener is typically set up when the DomainParticipant is created (see 9.3.1 Creating a DomainParticipant on page 615). You can also set one up after creation time by using the set_listener() operation, as illustrated in **Figure 9.5: Setting up DomainParticipantListener** below. The get_listener() operation can be used to retrieve the current DomainParticipantListener.

#### Figure 9.5: Setting up DomainParticipantListener

```cpp
// MyDomainParticipantListener only handles PUBLICATION_MATCHED and
// SUBSCRIPTION_MATCHED status for DomainParticipant Entities
class MyDomainParticipantListener : public DDSDomainParticipantListener {
  public:
    virtual void on_publication_matched(DDSDataWriter *writer,
                                         const DDS_PublishedMatchedStatus &status);
    virtual void on_subscription_matched(DDSDataReader *reader,
                                          const DDS_SubscribedMatchedStatus &status);
};
void MyDomainParticipantListener::on_publication_matched(
  DDSDataWriter *writer,
  const DDS_PublishedMatchedStatus &status)
{
  const char *name = writer->get_topic()->get_name();
  printf("Number of matching DataReaders for Topic %s is %d\n", name, status.current_count);
}
```
9.3.6 Setting Up DomainParticipantListeners

```c
};

void MyDomainParticipantListener::on_subscription_matched(
    DDSDataReader* reader,
    const DDS_SubscriptionMatchedStatus& status)
{
    const char* name =
        reader->get_topicdescription()->get_name();
    printf("Number of matching DataWriters for Topic %s is %d\n",
        name, status.current_count);
}
```

// Set up participant listener
MyDomainParticipantListener* participant_listener =
    new MyDomainParticipantListener();
if (participant_listener == NULL) {
    // ... handle error
}

// Create the participant with a listener
DDSDomainParticipant* participant = factory->create_participant(
    domain_id, participant_qos, participant_listener,
    DDS_PUBLICATION_MATCHED_STATUS |
    DDS_SUBSCRIPTION_MATCHED_STATUS);
if (participant == NULL) {
    // ... handle error
}
```

If a Listener is set for a DomainParticipant, the Listener needs to exist as long as the DomainParticipant exists. It is unsafe to destroy the Listener while it is attached to a participant. However, you may remove the DomainParticipantListener from a DomainParticipant by calling set_listener() with a NULL value. Once the Listener has been removed from the participant, you may safely destroy it (see 4.4.1 Types of Listeners on page 190).

Notes:

- Due to a thread-safety issue, the destruction of a DomainParticipantListener from an enabled DomainParticipant should be avoided—even if the DomainParticipantListener has been removed from the DomainParticipant. (This limitation does not affect the Java API.)
- It is possible for multiple internal Connext DDS threads to call the same method of a DomainParticipantListener simultaneously. You must write the methods of a DomainParticipantListener to be multithread safe and reentrant. The methods of the Listener of other Entities do not have this constraint and are guaranteed to have single threaded access.

See also:

- 5.1.5 Setting Up TopicListeners on page 223
- 7.2.5 Setting Up PublisherListeners on page 284
- 7.3.4 Setting Up DataWriterListeners on page 294
9.3.7 Setting DomainParticipant QosPolicies

A DomainParticipant’s QosPolicies are used to configure discovery, database sizing, threads, information sent to other DomainParticipants, and the behavior of the DomainParticipant when acting as a factory for other Entities.

Note: set_qos() cannot always be used in a listener callback; see 4.5.1 Restricted Operations in Listener Callbacks on page 200.

The DDS_DomainParticipantQos structure has the following format:

```c
struct DDS_DomainParticipantQos {
    DDS_UserDataQosPolicy user_data;
    DDS_EntityFactoryQosPolicy entity_factory;
    DDS_WireProtocolQosPolicy wire_protocol;
    DDS_TransportBuiltInQosPolicy transport_builtin;
    DDS_TransportUnicastQosPolicy default_unicast;
    DDS_DiscoveyQosPolicy discovery;
    DDS_DomainParticipantResourceLimitsQosPolicy resource_limits;
    DDS_EventQosPolicy event;
    DDS_ReceiverPoolQosPolicy receiver_pool;
    DDS_DatabaseQosPolicy database;
    DDS_DiscoveryConfigQosPolicy discovery_config;
    DDS_PropertyQosPolicy property;
    DDS_EntityNameQosPolicy participant_name;
    DDS_TransportMulticastMappingQosPolicy multicast_mapping;
    DDS_ServiceQosPolicy service;
    DDS_TypeSupportQosPolicy type_support;
};
```

Table 9.4 DomainParticipant QosPolicies summarizes the meaning of each policy (listed alphabetically). For information on why you would want to change a particular QosPolicy, see the section referenced in the table.

### Table 9.4 DomainParticipant QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database</td>
<td>Various settings and resource limits used by Connext DDS to control its internal database. See 9.5.1 DATABASE QosPolicy (DDS Extension) on page 637.</td>
</tr>
<tr>
<td>Discovery</td>
<td>Configures the mechanism used by Connext DDS to automatically discover and connect with new remote applications. See 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640.</td>
</tr>
<tr>
<td>DiscoveryConfig</td>
<td>Controls the amount of delay in discovering entities in the system and the amount of discovery traffic in the network. See 9.5.3 DISCOVERY_Config QosPolicy (DDS Extension) on page 644.</td>
</tr>
</tbody>
</table>
### Table 9.4 DomainParticipant QosPolicies

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainParticipantResourceLimits</td>
<td>Various settings that configure how DomainParticipants allocate and use physical memory for internal resources, including the maximum sizes of various properties. See 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654.</td>
</tr>
<tr>
<td>EntityFactory</td>
<td>Controls whether or not child entities are created in the enabled state. See 7.4.2 ENTITYFACTORY QosPolicy on page 348.</td>
</tr>
<tr>
<td>EntityName</td>
<td>Assigns a name to a DomainParticipant. See 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
<tr>
<td>Event</td>
<td>Configures the DomainParticipant's internal thread that handles timed events. See 9.5.5 EVENT QosPolicy (DDS Extension) on page 662.</td>
</tr>
<tr>
<td>Property</td>
<td>Stores name/value(string) pairs that can be used to configure certain parameters of Connext DDS that are not exposed through formal QoS policies. It can also be used to store and propagate application-specific name/value pairs, which can be retrieved by user code during discovery. See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.</td>
</tr>
<tr>
<td>ReceiverPool</td>
<td>Configures threads used by Connext DDS to receive and process data from transports (for example, UDP sockets). See 9.5.6 RECEIVER_POOL QosPolicy (DDS Extension) on page 664.</td>
</tr>
<tr>
<td>Service</td>
<td>Intended for use by RTI infrastructure services. User applications should not modify its value. See 7.5.23 SERVICE QosPolicy (DDS Extension) on page 454.</td>
</tr>
<tr>
<td>TransportBuiltin</td>
<td>Specifies which built-in transport plugins are used. See 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665.</td>
</tr>
<tr>
<td>TransportMulticastMapping</td>
<td>Specifies the automatic mapping between a list of topic expressions and multicast address that can be used by a DataReader to receive data for a specific topic. See 9.5.8 TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension) on page 667.</td>
</tr>
<tr>
<td>TransportUnicast</td>
<td>Specifies a subset of transports and port number that can be used by an Entity to receive data. See 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460.</td>
</tr>
<tr>
<td>TypeSupport</td>
<td>Used to attach application-specific value(s) to a DataWriter or DataReader. These values are passed to the serialization or deserialization routine of the associated data type. See 7.5.29 TYPESUPPORT QosPolicy (DDS Extension) on page 464.</td>
</tr>
<tr>
<td>UserData</td>
<td>Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to Connext DDS's discovery meta-data. See 7.5.30 USER_DATA QosPolicy on page 465.</td>
</tr>
<tr>
<td>WireProtocol</td>
<td>Specifies IDs used by the RTPS wire protocol to create globally unique identifiers. See 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670.</td>
</tr>
</tbody>
</table>

### 9.3.7.1 Configuring QoS Settings when DomainParticipant is Created

As described in 9.3.1 Creating a DomainParticipant on page 615, there are different ways to create a DomainParticipant, depending on how you want to specify its QoS (with or without a QoS Profile).
• **Figure 9.4:** Creating a `DomainParticipant` with Default QoS Policies on page 616 has an example of how to create a `DomainParticipant` with default QoS Policies by using the special constant, `DDS_PARTICIPANT_QOS_DEFAULT`, which indicates that the default QoS values for a `DomainParticipant` should be used. The default `DomainParticipant` QoS values are configured in the `DomainParticipantFactory`; you can change them with `set_default_participant_qos()` or `set_default_participant_qos_with_profile()` (see 9.2.2 Getting and Setting Default QoS for DomainParticipants on page 607). Then any `DomainParticipants` created with the `DomainParticipantFactory` will use the new default values. As described in 4.1.7 Getting, Setting, and Comparing QoS Policies on page 172, this is a general pattern that applies to the construction of all Entities.

• To create a `DomainParticipant` with non-default QoS without using a QoS Profile, see the example code in **Figure 9.6:** Creating `DomainParticipant` with Modified QoS Policies (not from profile) below. It uses the `DomainParticipantFactory`’s `get_default_participant_qos()` method to initialize a DDS_ParticipantQos structure. Then, the policies are modified from their default values before the structure is used in the `create_participant()` method.

• You can also create a `DomainParticipant` and specify its QoS settings via a QoS Profile. To do so, you will call `create_participant_with_profile()`, as seen in **Figure 9.7:** Creating `DomainParticipant` with QoS Profile on the next page.

• If you want to use a QoS profile, but then make some changes to the QoS before creating the `DomainParticipant`, call `get_participant_qos_from_profile()` and `create_participant()` as seen in **Figure 9.8:** Getting QoS from Profile, Creating `DomainParticipant` with Modified QoS Values on the next page.

For more information, see 9.3.1 Creating a `DomainParticipant` on page 615 and Configuring QoS with XML (Chapter 19 on page 848).

**Notes:**

• The examples in this section use the Traditional C++ API; for examples in the Modern C++ API, see the sections "Participant Use Cases," "QoS Use Cases," and "Qos Provider Use Cases" in the API Reference HTML documentation, under "Programming How-To's."

• In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QoS Policy Handling Considerations for C on page 182.

**Figure 9.6:** Creating `DomainParticipant` with Modified QoS Policies (not from profile)

```c
int main()
{
  DDS_DomainId_t domain_id = 10;
  DDS_DomainParticipantQos participant_qos;
  // initialize participant_qos with default values
  factory->get_default_participant_qos(participant_qos);
  // make QoS changes here
  participant_qos.wire_protocol.participant_id = 2;
  // Create the participant with modified qos
  DDS_DomainParticipant* participant = factory->create_participant(
```
domain_id, participant_qos, NULL, DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    // ... error
}

Figure 9.7: Creating DomainParticipant with QoS Profile

DDS_DomainId_t domain_id = 10;
// MyDomainParticipantListener is user defined and
// extends DDSDomainParticipantListener
MyDomainParticipantListener* participant_listener
    = new MyDomainParticipantListener(); // or = NULL
// Create the participant
DDSDomainParticipant* participant =
    factory->create_participant_with_profile(domain_id,
        "MyDomainLibrary", "MyDomainProfile",
        participant_listener, DDS_STATUS_MASK_ALL);
if (participant == NULL) {
    // ... error
};

Figure 9.8: Getting QoS from Profile, Creating DomainParticipant with Modified QoS Values

DDS_DomainParticipantQos participant_qos;
// Get DomainParticipant QoS from profile
retcode = factory->get_participant_qos_from_profile(participant_qos,
        "DomainParticipantProfileLibrary", "DomainParticipantProfile");
if (retcode != DDS_RETCODE_OK) {
    // handle error
}
// Makes QoS changes here
participant_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
// create participant with modified QoS
DDSDomainParticipant* participant = factory->create_participant(domain_id,
        participant_qos, NULL, DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    // handle error
}

9.3.7.2 Comparing QoS Values

The equals() operation compares two DomainParticipant’s DDS_DomainParticipantQos structures for equality. It takes two parameters for the two DomainParticipant’s QoS structures to be compared, then returns TRUE if they are equal (all values are the same) or FALSE if they are not equal.

9.3.7.3 Changing QoS Settings After DomainParticipant Has Been Created

There are two ways to change an existing DomainParticipant’s QoS after it is has been created—again depending on whether or not you are using a QoS Profile.

- To change QoS programmatically (that is, without using a QoS Profile), use get_qos() and set_qos(). See the example code in Figure 9.9: Changing QoS of Existing Participant (without QoS Profile) on the next page. It retrieves the current values by calling the DomainParticipant’s get_qos()
9.3.7 Setting DomainParticipant QoS Policies

operation. Then it modifies the value and calls `set_qos()` to apply the new value. Note, however, that some QoS policies cannot be changed after the DomainParticipant has been enabled—this restriction is noted in the descriptions of the individual QoS policies.

- You can also change a `DomainParticipant`’s (and all other Entities’) QoS by using a QoS Profile and calling `set_qos_with_profile()`. For an example, see Figure 9.10: Changing QoS of Existing Participant with QoS Profile below. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

Note:

- In C, you must initialize the QoS structures before they are used, see 4.2.2 Special QoS Policy Handling Considerations for C on page 182.

**Figure 9.9: Changing QoS of Existing Participant (without QoS Profile)**

```c
DDS_DomainParticipantQos participant_qos;
// Get current QoS
//participant points to an existing DDSDomainParticipant
if (participant->get_qos(participant_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Make QoS changes
participant_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
// Set the new QoS
if (participant->set_qos(participant_qos) != DDS_RETCODE_OK) {
    // handle error
}
```

**Figure 9.10: Changing QoS of Existing Participant with QoS Profile**

```c
DDS_DomainParticipantQos participant_qos;
// Get current QoS
//participant points to an existing DDSDomainParticipant
if (participant->get_qos(participant_qos) != DDS_RETCODE_OK) {
    // handle error
}
// Make QoS changes
participant_qos.entity_factory.autoenable_created_entities = DDS_BOOLEAN_FALSE;
// Set the new QoS
if (participant->set_qos(participant_qos) != DDS_RETCODE_OK) {
    // handle error
}
```

### 9.3.7.4 Getting and Setting DomainParticipant’s Default QoS Profile and Library

You can get the default QoS profile for the `DomainParticipant` with the `get_default_profile()` operation. You can also get the default library for the `DomainParticipant`, as well as the library that contains the `DomainParticipant`’s default profile (these are not necessarily the same library); these operations are called `get_default_library()` and `get_default_library_profile()`, respectively. These operations are for
9.3.7 Setting DomainParticipant QosPolicies

informational purposes only (that is, you do not need to use them as a precursor to setting a library or profile.) For more information, see Configuring QoS with XML (Chapter 19 on page 848).

```cpp
virtual const char * get_default_library ()
const char * get_default_profile ()
const char * get_default_profile_library ()
```

There are also operations for setting the DomainParticipant’s default library and profile:

```cpp
DDS_ReturnCode_t set_default_library ( const char * library_name)
DDS_ReturnCode_t set_default_profile ( const char * library_name, const char * profile_name)
```

If the default profile/library is not set, the DomainParticipant inherits the default from the DomainParticipantFactory.

- **set_default_profile()** specifies the profile that will be used as the default the next time a default DomainParticipant profile is needed during a call to one of this DomainParticipant’s operations. When calling a DomainParticipant operation that requires a profile_name parameter, you can use NULL to refer to the default profile. (This same information applies to setting a default library.)

- **set_default_profile()** does not set the default QoS for entities created by the DomainParticipant; for this functionality, use the DomainParticipant’s set_default_<entity>_qos_with_profile() operation (you may pass in NULL after having called set_default_profile(), see 9.3.7.5 Getting and Setting Default QoS for Child Entities below).

- **set_default_profile()** does not set the default QoS for newly created DomainParticipants; for this functionality, use the DomainParticipantFactory’s set_default_participant_qos_with_profile(), see 9.2.2 Getting and Setting Default QoS for DomainParticipants on page 607).

### 9.3.7.5 Getting and Setting Default QoS for Child Entities

The set_default_<entity>_qos() and set_default_<entity>_qos_with_profile() operations set the default QoS that will be used for newly created entities (where <entity> may be publisher, subscriber, datawriter, datareader, or topic). The new QoS settings will only be used if DDS_<entity>_QOS_DEFAULT is specified as the qos parameter when create_<entity>() is called. For example, for a Publisher, you can use either:

```cpp
DDS_ReturnCode_t set_default_publisher_qos ( const DDS_PublisherQos &qos)
DDS_ReturnCode_t set_default_publisher_qos_with_profile ( const char *library_name, const char *profile_name)
```

The following operation gets the default QoS that will be used for creating Publishers if DDS_PUBLISHER_QOS_DEFAULT is specified as the ‘qos’ parameter when create_publisher() is called:
There are similar operations for Subscribers, Data Writers, Data Readers and Topics. These operations, get default \_<entity>\_qos(), get the QoS settings that were specified on the last successful call to set default \_<entity>\_qos() or set default \_<entity>\_qos with profile(), or if the call was never made, the default values listed in DDS \_<entity>\_Qos. They may potentially allocate memory depending on the sequences contained in some QoS policies.

**Note:** It is not safe to set default QoS values for an entity while another thread may be simultaneously getting or setting them, or using the QOS_DEFAULT constant to create the entity.

### 9.3.8 Looking up Topic Descriptions

The lookup topicdescription() operation allows you to access a locally created DDSTopicDescription based on the Topic’s name.

```c
DDSTopicDescription* lookup_topicdescription(const char *topic_name)
```

**DDSTopicDescription** is the base class for Topics, MultiTopics\(^1\) and ContentFilteredTopics. You can narrow the DDSTopicDescription returned from lookup_topicdescription() to a Topic or ContentFilteredTopic as appropriate.

Unlike find_topic() (see 9.3.9 Finding a Topic below), which logically returns a new Topic that must be independently deleted, this operation returns a reference to the original local object.

If no TopicDescription has been created yet with the given Topic name, this method will return a NULL value.

The DomainParticipant does not have to be enabled when you call lookup_topicdescription().

**Note:** It is not safe to create or delete a topic while another thread is calling lookup_topicdescription() for that same topic.

### 9.3.9 Finding a Topic

The find_topic() operation finds an existing (or ready to exist) Topic, based on its name. This call can be used to block for a specified duration to wait for the Topic to be created.

```c
DDSTopic* DDSDomainParticipant::find_topic (const char * topic_name,
                   const DDS_Duration_t & timeout)
```

If the requested Topic already exists, it is returned. Otherwise, find_topic() waits until either another thread creates it, or returns when the specified timeout occurs.

find_topic() is useful when multiple threads are concurrently creating and looking up topics. In that case, one thread can call find_topic() and, if another thread has not yet created the topic being looked up, it can

\(^1\)Multitopics are not supported.
wait for some period of time for it to do so. In almost all other cases, it is more straightforward to call `lookup_topicdescription()` (see 9.3.8 Looking up Topic Descriptions on the previous page).

The DomainParticipant must be enabled when you call `find_topic()`.

**Note:** Each DDSTopic obtained by `find_topic()` must also be deleted by calling the DomainParticipant’s `delete_topic()` operation (see 5.1.2 Deleting Topics on page 219).

### 9.3.10 Getting the Implicit Publisher or Subscriber

The `get_implicit_publisher()` operation allows you to access the DomainParticipant’s implicit Publisher. If one does not already exist, this operation creates an implicit Publisher.

There is a similar operation for implicit Subscribers:

```c
DDSPublisher * get_implicit_publisher();
DDSSubscriber * get_implicit_subscriber();
```

There can only be one implicit Publisher and one implicit Subscriber per DomainParticipant. They are created with default QoS values (DDS_PUBLISHER_QOS_DEFAULT) and no Listener. For more information, see 7.2.1 Creating Publishers Explicitly vs. Implicitly on page 275. You can use an implicit Publisher or implicit Subscriber just like an explicitly created one.

An implicit Publisher/Subscriber is deleted automatically when `delete_contained_entities()` is called. It can also be deleted by calling `delete_publisher/subscriber()` with the implicit Publisher/Subscriber as a parameter.

When a DomainParticipant is deleted, if there are no attached DataReaders that belong to the implicit Subscriber or no attached DataWriters that belong to the implicit Publisher, any implicit Publisher/Subscriber will be deleted by the middleware implicitly.

**Note:** It is not safe to create an implicit Publisher/Subscriber while another thread may be simultaneously calling `set_default_[publisher/subscriber]_qos()`.

**How to get the implicit Publisher/Subscriber. (For simplicity, error handling is not shown.)**

```c
using namespace DDS;
...
Publisher * publisher = NULL;
Subscriber * subscriber = NULL;
PublisherQos publisher_qos;
SubscriberQos subscriber_qos;
...
publisher = participant->get_implicit_publisher();
/* Change implicit publisher QoS */
publisher->get_qos(publisher_qos);
publisher_qos.partition.name.maximum(3);
publisher_qos.partition.name.length(3);
publisher_qos.partition.name[0] = DDS_String_dup("partition_A");
publisher_qos.partition.name[1] = DDS_String_dup("partition_B");
publisher_qos.partition.name[2] = DDS_String_dup("partition_C");
```

629
9.3.11 Asserting Liveliness

The `assert_liveliness()` operation manually asserts the liveliness of all the `DataWriters` created by this `DomainParticipant` that has 7.5.15 LIVELINESS QosPolicy on page 427 kind set to MANUAL_BY_PARTICIPANT. When `assert_liveliness()` is called, then for those `DataWriters` who have their LIVELINESS set to MANUAL_BY_PARTICIPANT, Connext DDS will send a packet to all matched `DataReaders` that indicates that the `DataWriter` is still alive.

However, the LIVELINESS contract of periodically sending liveliness packets to `DataReaders` is also fulfilled when the `write()`, `assert_liveliness()`, `unregister_instance()` and `dispose()` operations on a `DataWriter` itself is called. Those calls will also cause Connext DDS to send packets that indicate the liveliness of the `DataWriter`. Therefore, it is necessary for the application to call `assert_liveliness()` on the `DomainParticipant` only if those operations on a `DataWriter` are not being invoked within the period specified by the 7.5.15 LIVELINESS QosPolicy on page 427.

9.3.12 Learning about Discovered DomainParticipants

The `get_discovered_participants()` operation provides you with a list of `DomainParticipants` that have been discovered in the DDS domain (except any that you have said to ignore via the `ignore_participant()` operation (see 18.4 Restricting Communication—Ignoring Entities on page 841)).

Once you have a list of discovered `DomainParticipants`, you can get more information about them by calling the `get_discovered_participant_data()` operation. This operation can only be used on `DomainParticipants` that are in the same DDS domain and have not been marked as ‘ignored.’ Otherwise, the operation will fail and return DDS_RET_CODE_PRECONDITION_NOT_MET. The returned information is of type DDS_ParticipantBuiltInTopicData, described in Table 18.1 Participant Built-in Topic’s Data Type (DDS_ParticipantBuiltInTopicData).

9.3.13 Learning about Discovered Topics

The `get_discovered_topics()` operation provides you with a list of `Topics` that have been discovered in the DDS domain (except any that you have said to ignore via the `ignore_topic()` operation (see 18.4 Restricting Communication—Ignoring Entities on page 841)).

Once you have a list of discovered `Topics`, you can get more information about them by calling the `get_discovered_topic_data()` operation. This operation can only be used on `Topics` that have been created by a
DomainParticipant in the same DDS domain as the participant on which this operation is invoked and must not have been "ignored" by means of the DomainParticipant ignore_topic() operation. Otherwise, the operation will fail and return DDS_RETCODE_PRECONDITION_NOT_MET. The returned information is of type DDS_TopicBuiltinTopicData, described in Table 18.4 Topic Built-in Topic’s Data Type (DDS_TopicBuiltinTopicData).

9.3.14 Getting Participant Protocol Status

Statistics about corrupted RTPS messages received by the participant can be obtained from the DomainParticipantProtocolStatus.

Table 9.5 DDS_DomainParticipantProtocolStatus

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LongLong</td>
<td>corrupted_rtps_message_count</td>
<td>The number of corrupted RTPS messages detected by the participant.</td>
</tr>
<tr>
<td>DDS_LongLong</td>
<td>corrupted_rtps_message_count_change</td>
<td>The incremental change in the number of corrupted messages detected since the last time the status was read.</td>
</tr>
<tr>
<td>DDS_Time_t</td>
<td>last_corrupted_message_timestamp</td>
<td>The timestamp of the last corrupted RTPS message detected by the participant.</td>
</tr>
</tbody>
</table>

You can get the DomainParticipantProtocolStatus by using the get_participant_protocol_status() operation. The WireProtocolQosPolicy’s compute_crc and check_crc must be enabled in the publishing and subscribing applications, respectively, when the protocol status is obtained.

9.3.15 Other DomainParticipant Operations

9.3.15.1 Verifying Entity Containment

If you have a handle to an Entity, and want to see if that Entity was created from your DomainParticipant (or any of its Publishers or Subscribers), use the contains_entity() operation, which returns a boolean.

An Entity’s instance handle may be obtained from built-in topic data (see Built-In Topics (Chapter 18 on page 831)), various statuses, or from the get_instance_handle() operation (see 4.1.3 Getting an Entity’s Instance Handle on page 171).

9.3.15.2 Getting the Current Time

The get_current_time() operation returns the current time value from the same time-source (clock) that Connext DDS uses to timestamp the data published by DataWriters (source_timestamp of the SampleInfo structure, see 8.4.6 The SampleInfo Structure on page 565). The time-sources used by Connext DDS do not have to be synchronized nor are they synchronized by Connext DDS.
9.4 DomainParticipantFactory QosPolicies

See also: 9.6 Clock Selection on page 677.

9.3.15.3 Getting All Publishers and Subscribers

The get_publishers() and get_subscribers() operations will provide you with a list of the DomainParticipant's Publishers and Subscribers, respectively.

9.4 DomainParticipantFactory QosPolicies

This section describes QosPolicies that are strictly for the DomainParticipantFactory (not the DomainParticipant). For a complete list of QosPolicies that apply to DomainParticipantFactory, see Table 9.2 DomainParticipantFactory QoS.

- 9.4.1 LOGGING QosPolicy (DDS Extension) below
- 9.4.2 PROFILE QosPolicy (DDS Extension) on the next page
- 9.4.3 SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension) on page 635

9.4.1 LOGGING QosPolicy (DDS Extension)

This QosPolicy configures the properties associated with the Connext DDS logging facility.

This QosPolicy includes the members in Table 9.6 DDS_LoggingQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

See also: 25.2 Controlling Messages from Connext DDS on page 984 and 25.2.2 Configuring Logging via XML on page 992.

Table 9.6 DDS_LoggingQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NDDS_ConfigLogVerbosity</td>
<td>verbosity</td>
<td>Specifies the verbosity at which Connext DDS diagnostic information will be logged.</td>
</tr>
<tr>
<td>NDDS_Config_LogCategory</td>
<td>category</td>
<td>Specifies the category for which logging needs to be enabled.</td>
</tr>
<tr>
<td>NDDS_Config_LogPrintfFormat</td>
<td>print_format</td>
<td>Specifies the format to be used to output Connext DDS diagnostic information.</td>
</tr>
<tr>
<td>char*</td>
<td>output_file</td>
<td>Specifies the file to which the logged output is redirected.</td>
</tr>
<tr>
<td>char*</td>
<td>output_file_suffix</td>
<td>Sets the file suffix when logging to a set of files.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_bytes_per_file</td>
<td>Specifies the maximum number of bytes a single file can contain.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_files</td>
<td>Specifies the maximum number of files to create before overwriting the previous ones.</td>
</tr>
</tbody>
</table>
9.4.1.1 Example

```cpp
DSDDomainParticipantFactory *factory =
    DSDDomainParticipantFactory::get_instance();
DDS_DomainParticipantFactoryQos factoryQos;
DDS_ReturnCode_t retcode = factory->get_qos(factoryQos);
if (retcode != DDS_RETCODE_OK) {
    // error
}
factoryQos.logging.output_file = DDS_String_dup("myOutput.txt");
factoryQos.logging.verbosity = NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL;
factory->set_qos(factoryQos);
```

9.4.1.2 Properties

This QosPolicy can be changed at any time.

Since it is only configuring logging, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

9.4.1.3 Related QosPolicies

- None

9.4.1.4 Applicable DDS Entities

- 9.2 DomainParticipantFactory on page 602

9.4.1.5 System Resource Considerations

Because the `output_file` will be freed by Connext DDS, you should use `DDS_String_dup()` to allocate the string when providing an `output_file`.

9.4.2 PROFILE QosPolicy (DDS Extension)

This QosPolicy determines the way that XML documents containing QoS profiles are loaded.

All QoS values for Entities can be configured with QoS profiles defined in XML documents. XML documents can be passed to Connext DDS in string form, or more likely, through files found on a file system. This QoS configures how a DomainParticipantFactory loads the QoS profiles defined in XML. QoS profiles may be stored in this QoS as XML documents as a string. The location of XML files defining QoS profiles may be configured via this QoS. There are also default locations where the DomainParticipantFactory will look for files to load QoS profiles. You may disable any or all of these default locations using the Profile QoS. For more information about QoS profiles and libraries, please see Configuring QoS with XML (Chapter 19 on page 848).

This QosPolicy includes the members in Table 9.7 DDS_ProfileQosPolicy. For the defaults and valid ranges, please refer to the API Reference HTML documentation.
### Table 9.7 DDS_ProfileQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_</td>
<td>string_profile</td>
<td>Sequence of strings (empty by default) containing an XML document to load.</td>
</tr>
<tr>
<td>StringSeq</td>
<td></td>
<td>The concatenation of the strings in this sequence must be a valid XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td>document according to the XML QoS profile schema.</td>
</tr>
<tr>
<td>url_profile</td>
<td></td>
<td>A sequence of XML files (empty by default) containing a set of XML</td>
</tr>
<tr>
<td></td>
<td></td>
<td>documents to load. See 19.5 How to Load XML-Specified QoS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Settings on page 882.</td>
</tr>
<tr>
<td>DDS_</td>
<td>ignore_user_profile</td>
<td>When TRUE, the QoS profiles contained in the file USER_QOS_PROFILES.xml</td>
</tr>
<tr>
<td>Boolean</td>
<td></td>
<td>in the current working directory will be ignored.</td>
</tr>
<tr>
<td></td>
<td>ignore_environment_profile</td>
<td>When TRUE, the value of the environment variable NDDS_QOS_PROFILES will</td>
</tr>
<tr>
<td></td>
<td></td>
<td>be ignored.</td>
</tr>
<tr>
<td></td>
<td>ignore_resource_profile</td>
<td>When TRUE, the QoS profiles in the file $NDDSHOME/resource/xml/NDDS_QOS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PROFILES.xml will be ignored.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NDDS_QOS_PROFILES.xml does not exist by default. However,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NDDS_QOS_PROFILES.example.xml is shipped with the host bundle of the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>product; you can copy it to NDDS_QOS_PROFILES.xml and modify it for your</td>
</tr>
<tr>
<td></td>
<td></td>
<td>own use.</td>
</tr>
</tbody>
</table>

In the Modern C++ API, there is not a PROFILE QosPolicy, because the class that manages QoS profiles is dds::core::QosProvider—not the DomainParticipantFactory. A QosProvider can receive a QosProviderParams instance, which encapsulates the fields described before.

#### 9.4.2.1 Example

**Traditional C++:**

```cpp
DDSDomainParticipantFactory *factory =
    DDSDomainParticipantFactory::get_instance();
DDS_DomainParticipantFactoryQos factoryQos;

DDS_ReturnCode_t retcode = factory->get_qos(factoryQos);
if (retcode != DDS_RETCODE_OK) {
    // error
}
const char *url_profiles[2] = {
    "file://usr/local/default_dds.xml",
    "file://usr/local/alternative_default_dds.xml"};
factoryQos.profile.url_profile.from_array(url_profiles, 2);

factoryQos.profile.ignore_resource_profile = DDS_BOOLEAN_TRUE;
factory->set_qos(factoryQos);
```

**Modern C++:**

```cpp
rti::core::QosProviderParams params =
    dds::core::QosProvider::Default() -> default_provider_params();

std::vector<std::string> url_profiles = {
```
9.4.3 SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension)

```cpp
params.url_profile(url_profiles);
params.ignore_resource_profile(true);

dds::core::QosProvider::Default()->default_provider_params(params);
```

9.4.2.2 Properties

This QosPolicy can be changed at any time.

Since it is only for the DomainParticipantFactory, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

9.4.2.3 Related QosPolicies

- None

9.4.2.4 Applicable Entities

- 9.2 DomainParticipantFactory on page 602

9.4.2.5 System Resource Considerations

Once the QoS profiles are loaded, the DomainParticipantFactory will keep one copy of each QoS in the QoS profiles in memory.

You can free the memory associated with the XML QoS profiles by calling the DomainParticipantFactory’s `unload_profiles()` operation.

9.4.3 SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension)

The SYSTEM_RESOURCE_LIMITS QosPolicy configures DomainParticipant-independent resources used by Connext DDS. Its main use is to change the maximum number of DomainParticipants that can be created within a single process (address space).

It contains the single member as shown in Table 9.8 DDS_SystemResourceLimitsQosPolicy. For the default and valid range, please refer to the API Reference HTML documentation.

### Table 9.8 DDS_SystemResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_objects_per_thread</td>
<td>Sizes the thread storage that is allocated on a per-thread basis when the thread calls Connext DDS APIs.</td>
</tr>
</tbody>
</table>

The only parameter that you can set, `max_objects_per_thread`, controls the size of thread-specific storage that is allocated by Connext DDS for every thread that invokes a Connext DDS API. This storage is used
to cache objects that have to be created on a per-thread basis when a thread traverses different portions of Connext DDS internal code.

Thus instead of dynamically creating and destroying the objects as a thread enters and leaves different parts of the code, Connext DDS caches the objects by storing them in thread-specific storage. We assume that a thread will repeatedly call Connext DDS APIs so that the objects cached will be needed again and again.

The number of objects that will be stored in the cache depends on the number of APIs (sections of Connext DDS code) that a thread invokes. It also depends on the number of different DomainParticipants with which the thread interacts. For a single DomainParticipant, the maximum number of objects that could be stored is a constant—indeed of the number of Entities created in or by the participant. A safe number to use is 200 objects per DomainParticipant.

A user thread that only interacts with a single DomainParticipant or the Entities thereof, would never have more than 200 objects stored in its cache. However, if the same thread invokes Connext DDS APIs on other Entities of other DomainParticipants, the maximum number of objects that may be stored will increase with the number of participants involved.

The default setting of this resource should work for most user applications. However, if your application uses more than 4 DomainParticipants, you may need to increase the value of max_objects_per_thread.

9.4.3.1 Properties

This QoS policy cannot be modified after the DomainParticipantFactory is used to create the first DomainParticipant or WaitSet in an application. Therefore, it cannot be set in an XML file—only in code.

This QoS can be set to different values in different applications without affecting interoperability.

9.4.3.2 Related QoS Policies

There are no interactions with other QosPolicies.

9.4.3.3 Applicable DDS Entities

- 9.2 DomainParticipantFactory on page 602

9.4.3.4 System Resource Considerations

max_objects_per_thread is used to determine the size of an array of pointers to objects used in a thread. Increasing max_objects_per_thread will increase the amount of memory allocated by Connext DDS for every thread that access Connext DDS code. This includes internal Connext DDS threads as well as user threads.
9.5 DomainParticipant QosPolicies

This section describes the QosPolicies that are strictly for DomainParticipants (and no other types of Entities). For a complete list of QosPolicies that apply to DomainParticipant, see Table 9.4 DomainParticipant QosPolicies.

- 9.5.1 DATABASE QosPolicy (DDS Extension) below
- 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640
- 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644
- 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654
- 9.5.5 EVENT QosPolicy (DDS Extension) on page 662
- 9.5.6 RECEIVER_POOL QosPolicy (DDS Extension) on page 664
- 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665
- 9.5.8 TRANSPORT_MULTIICAST_MAPPING QosPolicy (DDS Extension) on page 667
- 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670

9.5.1 DATABASE QosPolicy (DDS Extension)

The Database QosPolicy configures how Connext DDS manages its internal database, including how often it cleans up, the priority of the database thread, and limits on resources that may be allocated by the database. RTI uses an internal in-memory database to store information about entities created locally as well as remote entities found during the discovery process. This database uses a background thread to garbage-collect records related to deleted entities. When the DomainParticipant that maintains this database is deleted, it shuts down this thread.

It includes the members in Table 9.9 DDS_DatabaseQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

Table 9.9 DDS_DatabaseQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_ThreadSettings_t</td>
<td>thread.mask</td>
<td>Thread settings for the database thread used by Connext DDS to periodically remove deleted records from the database. The values used for these settings are OS-dependent; see the RTI Connext DDS Core Libraries Platform Notes for details. Note: thread.cpu_list and thread.cpu_rotation are not relevant in this QoS policy.</td>
</tr>
<tr>
<td></td>
<td>thread.priority</td>
<td></td>
</tr>
<tr>
<td></td>
<td>thread.stack_size</td>
<td></td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>shutdown_timeout</td>
<td>The maximum time that the DomainParticipant will wait for the database thread to terminate when the participant is destroyed.</td>
</tr>
</tbody>
</table>
### Table 9.9 DDS_DatabaseQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>cleanup_period</td>
<td>The period at which the database thread wakes up to removed deleted records.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>shutdown_cleanup_period</td>
<td>The period at which the database thread wakes up to removed deleted records when the DomainParticipant is being destroyed.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_records</td>
<td>The number of records that is initially created for the database. These records hold information for both local and remote entities that are dynamically created or discovered.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_skilist_level</td>
<td>This is a performance tuning parameter that optimizes the time it takes to search the database for a record. A 'Skip List' is an algorithm for maintaining a list that is faster to search than a binary tree. This value should be set to $\log_2(N)$, where $N$ is the maximum number of elements that will be stored in a single list. The list that stores the records for remote DataReaders or the one for remote DataWriters tend to have the most entries. So, the number of DataWriters or DataReaders in a system across all DomainParticipants in a single DDS domain, which ever is greater, can be used to set this parameter.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_weak_references</td>
<td>This parameter sets the maximum number of entries in the weak reference table. Weak references are used as a technique for ensuring that unreferenced objects are deleted. The actual number of weak references is permitted to grow from the value set by initial_weak_references to this maximum. To prevent Connext DDS from allocating memory for weak references after initialization, you should set the initial and maximum weak references to the same value. However, it is difficult to calculate how many weak references an application will use. To allow Connext DDS to grow the weak reference table as needed, and thus dynamically allocate memory, you should set the value of this field to DDS_LENGTH_UNLIMITED, the default setting.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_weak_references</td>
<td>The initial number of entries in the weak reference table. See max_weak_references. Connext DDS may decide to use a larger initial value if initial_weak_references is set too small. If you access this parameter after a DomainParticipant has been created, you will see the actual value used.</td>
</tr>
</tbody>
</table>

You may be interested in modifying the **shutdown_timeout** and **shutdown_cleanup_period** parameters to decrease the time it takes to delete a DomainParticipant when your application is shutting down.

The 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMTS QosPolicy (DDS Extension) on page 654 controls the memory allocation for elements stored in the database.

Real-time programmers will probably want to adjust the priorities of all of the threads created by Connext DDS relative to each other as well as relative to non-Connext DDS threads in their applications. Connext DDS Threading Model (Chapter 21 on page 908), 9.5.5 EVENT QosPolicy (DDS Extension) on page 662, and 9.5.6 RECEIVER_POOL QosPolicy (DDS Extension) on page 664 discuss the other threads that are created by Connext DDS.

A record in the database can be deleted only when no threads are using it. Connext DDS uses a thread that periodically checks the database if records that have been marked for deletion can be removed. This period is set by **cleanup_period**. When a DomainParticipant is being destroyed, the thread will wake up faster at
the `shutdown_cleanup_period` as other threads delete and release records in preparation for shutting down.

On Windows and VxWorks® systems, the thread that is destroying the `DomainParticipant` may block up to `shutdown_timeout` seconds while waiting for the database thread to finish removing all records and terminating. On other operating systems, the thread destroying the `DomainParticipant` will block as long as required for the database thread to terminate.

The default values for those and the rest of the parameters in this QosPolicy should be sufficient for most applications.

### 9.5.1.1 Example

The priority of the database thread should be set to the lowest priority among all threads in a real-time system. Although, the database thread should not be permitted to starve, the work that it performs is non-time-critical.

### 9.5.1.2 Properties

This QosPolicy cannot be modified after the `DomainParticipant` is created.

It can be set differently on the publishing and subscribing sides.

### 9.5.1.3 Related QosPolicies

- 9.5.4 `DOMAIN_PARTICIPANT RESOURCE LIMITS QosPolicy (DDS Extension)` on page 654
- 9.5.5 `EVENT QosPolicy (DDS Extension)` on page 662
- 9.5.6 `RECEIVER_POOL QosPolicy (DDS Extension)` on page 664

### 9.5.1.4 Applicable DDS Entities

- 9.3 `DomainParticipants` on page 609

### 9.5.1.5 System Resource Considerations

Setting the thread parameters correctly on a real-time operating system is usually critical to the proper overall functionality of the applications on that system. Larger values for the `thread.stack_size` parameter will use up more memory.

Smaller values for the `cleanup_period` and `shutdown_cleanup_period` will cause the database thread to wake up more frequently using more CPU.

`Connext DDS` is permitted to use up more memory for larger values of `max_skiplist_level` and `max_weak_references`. Whether or not more memory is actually used depends on actual operating conditions.
9.5.2 DISCOVERY QosPolicy (DDS Extension)

The DISCOVERY QoS configures how DomainParticipants discover each other on the network. It identifies where on the network this application can potentially discover other applications with which to communicate. The middleware will periodically send network packets to these locations, announcing itself to any remote applications that may be present, and will listen for announcements from those applications. The discovery process is described in detail in Discovery (Chapter 15 on page 764).

This QosPolicy includes the members in Table 9.10 DDS_DiscoveryQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

### Table 9.10 DDS_DiscoveryQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_FloatSeq</td>
<td>enabled_transports</td>
<td>Transports available for use by the discovery process. See 9.5.2.1 Transports Used for Discovery below.</td>
</tr>
<tr>
<td>DDS_FloatSeq</td>
<td>initial_peers</td>
<td>Unicast locators (address/indices) of potential participants with which this DomainParticipant will attempt to establish communications. See 9.5.2.2 Setting the ‘Initial Peers’ List below.</td>
</tr>
<tr>
<td>DDS_FloatSeq</td>
<td>multicast_received_addresses</td>
<td>List of multicast addresses on which Discovery-related messages can be received by the DomainParticipant. See 9.5.2.4 Configuring Multicast Receive Addresses on page 642.</td>
</tr>
<tr>
<td>DDS_Byte</td>
<td>metatraffic_transport_priority</td>
<td>Transport priority to be used for sending Discovery messages. See 9.5.2.5 Meta-Traffic Transport Priority on page 642.</td>
</tr>
<tr>
<td>DDS_Byte</td>
<td>accept_unknown_peers</td>
<td>Whether to accept a participant discovered via unicast that is not in the initial_peers list. See 9.5.2.6 Controlling Acceptance of Unknown Peers on page 642.</td>
</tr>
<tr>
<td>DDS_Byte</td>
<td>enable_endpoint_discovery</td>
<td>Whether endpoint discovery will automatically occur with discovered DomainParticipants. See 18.4.5 Supervising Endpoint Discovery on page 845.</td>
</tr>
</tbody>
</table>

9.5.2.1 Transports Used for Discovery

The enabled_transports field allows you to specify the set of installed and enabled transports that can be used to discover other DomainParticipants. This field is a sequence of strings where each string specifies an alias of a registered (and thus installed and enabled) transport. Please see the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Pluggable Transports) for more information.

9.5.2.2 Setting the ‘Initial Peers’ List

When a DomainParticipant is created, it needs to find other participants in the same DDS domain—this is known as the ‘discovery process’ which is discussed in Discovery (Chapter 15 on page 764). One way to do so is to use this QosPolicy to specify a list of potential participants. This is the role of the parameter initial_peers. The strings containing peer descriptors are stored in the initial_peers string sequence. The format of a string discussed in 15.2.1 Peer Descriptor Format on page 769.
The peers stored in initial_peers are merely potential peers—there is no requirement that the peer DomainParticipants are actually up and running or even will eventually exist. The Connext DDS discovery process will try to contact all potential peer participants in the list periodically using unicast transports (as configured by the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644).

The initial_peers parameter can be modified in source code or it can be initialized from an environment variable, NDDS_DISCOVERY_PEERS or from a text file, see 15.2 Configuring the Peers List Used in Discovery on page 767.

Some transports accept the use of hostnames to specify the initial peers. By default, Connext DDS only resolves those hostnames when the DomainParticipant is created, but the DNS tracker can be used to keep the IP addresses of these hostnames updated. See 17.3 Using DNS Tracker to Keep Peer List Updated on page 830.

9.5.2.3 Adding and Removing Peers List Entries

The DomainParticipant’s add_peer() operation adds a peer description to the internal peer list that was initialized by the initial_peers field of the DISCOVERY QosPolicy.

```c
DDS_ReturnCode_t DDSDomainParticipant::add_peer (    const char* peer_desc)
```

The peer_desc string must be formatted as specified in 15.2.1 Peer Descriptor Format on page 769.

You can call this operation any time after the DomainParticipant has been enabled. An attempt will be made to contact the new peer immediately.

Adding peers with this operation has no effect on the initial_peers list. After a DomainParticipant has been created, the contents of the initial_peers field merely shows what the internal peer list was initialized to be. Therefore, initial_peers may not reflect the actual potential peer list used by a DomainParticipant. Furthermore, if you call get_qos(), the returned list of peers will not include the added peer—get_qos() will only show you what is set in the initial_peers list.

A peer added with add_peer() is not considered to be “unknown.” (That is, you may have accept_unknown_peers (9.5.2.6 Controlling Acceptance of Unknown Peers on the next page) set to FALSE and still use add_peer().)

This behavior may change depending on the DNS tracker configuration; see 17.3 Using DNS Tracker to Keep Peer List Updated on page 830. Adding a peer with a hostname that cannot be resolved into an IP address will produce an error if the DNS Tracker has not been enabled for that DomainParticipant.

You can remove an entry from the list with remove_peer(). Note that remove_peer() is only supported if Simple Participant Discovery (see 15.1.1 Simple Participant Discovery on page 765) is enabled for the Participant.

You can ignore data from a participant by using the ignore_participant() operation described in 18.4 Restricting Communication—Ignoring Entities on page 841.
9.5.2.4 Configuring Multicast Receive Addresses

The `multicast_receive_addresses` field in the DISCOVERY QosPolicy is a sequence of strings that specifies a set of multicast group addresses on which the `DomainParticipant` will listen for discovery meta-traffic. Each string must have a valid multicast address in either IPv4 dot notation or IPv6 presentation format. Please look at publicly available documentation of the IPv4 and IPv6 standards for the definition and valid address ranges for multicast.

The `multicast_receive_addresses` field can be initialized from multicast addresses that appear in the `NDDS_DISCOVERY_PEERS` environment variable or text file, see 15.2 Configuring the Peers List Used in Discovery on page 767. A multicast address found in the environment variable or text file will be added both to the `initial_peers` and `multicast_receive_addresses` fields. Note that the addresses in `initial_peers` are ones in which the `DomainParticipant` will send discovery meta-traffic, and the ones in `multicast_receive_addresses` are used for receiving discovery meta-traffic.

If `NDDS_DISCOVERY_PEERS` does not contain a multicast address, then `multicast_receive_addresses` is cleared and the RTI discovery process will not listen for discovery messages via multicast.

If `NDDS_DISCOVERY_PEERS` contains one or more multicast addresses, the addresses are stored in `multicast_receive_addresses`, starting at element 0. They will be stored in the order in which they appear in `NDDS_DISCOVERY_PEERS`.

**Note:** Currently, Connext DDS will only listen for discovery traffic on the first multicast address (element 0) in `multicast_receive_addresses`.

If you want to send discovery meta-traffic on a different set of multicast addresses than you want to receive discovery meta-traffic, set `initial_peers` and `multicast_receive_addresses` via the QosPolicy API.

9.5.2.5 Meta-Traffic Transport Priority

The `metatraffic_transprt_priority` field is used to specify the transport priority to be used for sending all discovery meta-traffic. See the 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458 for details on how transport priorities may be used.

9.5.2.6 Controlling Acceptance of Unknown Peers

The `accept_unknown_peers` field controls whether or not a `DomainParticipant` is allowed to communicate with other `DomainParticipants` found via unicast transport that are not in its peers list (which is the combination of the `initial_peers` list and any peers added with the `add_peer()` operation described in 9.5.2.3 Adding and Removing Peers List Entries on the previous page).

Suppose Participant A is included in Participant B’s initial peers list, but Participant B is not in Participant A’s list. When Participant B contacts Participant A by sending it a unicast discovery packet, then Participant A has a choice:
9.5.2 DISCOVERY QosPolicy (DDS Extension)

- If `accept_unknown_peers` is DDS_BOOLEAN_TRUE, then Participant A will reply to Participant B, and communications will be established.

- If `accept_unknown_peers` is DDS_BOOLEAN_FALSE, then Participant A will ignore Participant B, and A and B will never talk.

Note that Participants do not exchange peer lists. So if Participant A knows about Participant B, and Participant B knows about Participant C, Participant A will not discover Participant C.

**Note:** If `accept_unknown_peers` is false and shared memory is disabled, applications on the same node will not communicate if only ‘localhost’ is specified in the peer list. If shared memory is disabled or ‘shmem://’ is not specified in the peer list, if you want to communicate with other applications on the same node through the loopback interface, you must put the actual node address or hostname in NDDS_DISCOVERY_PEERS.

9.5.2.7 Example

You will always use this policy to set the participant_id when you want to run more than one DomainParticipant in the same DDS domain on the same host.

The easiest way to set the initial peers list is to use the NDDS_DISCOVERY_PEERS environment variable. However, should you want asymmetric multicast addresses for sending or receiving meta-traffic, you will need to use this QosPolicy directly.

A reason to use asymmetric multicast addresses is to take advantage of the efficiency provided by using multicast, while at the same time preventing all participants from discovering each other. For example, suppose you have a system in which you have a single server node and a hundred client nodes. The client nodes do not publish or subscribe to each other’s data and thus never need to know about each others existence.

If we did not use multicast, we would have to populate the server application’s peer list with 100 peer descriptors for each of the client nodes. Each client application would only need to have the server application in its peer list. The maintenance of the list is unwieldy, especially if nodes are constantly reconfigured and addresses changed. In addition, the server will send out discovery packets on a per client basis since the peer list essentially holds 100 unicast addresses.

Instead, if we used a single multicast address in the NDDS_DISCOVERY_PEERS environment variable, the server and all of the clients would discover each other. Certainly, the list is easier to maintain, but the total amount of traffic has actually increased since the clients are now exchanging packets with each other uselessly.

To keep the list maintainable, as well as to minimize discovery traffic, we can have the server send out packets on a multicast address by modifying its initial_peer field. The clients would have their multicast_receive_addresses field set to the same address used by the server. The initial_peers of the clients would only need the single unicast peer descriptor of the server as before.
Now, the server can send a single packet that will be received by all of the clients, but the clients will not discover each other because they never send out a multicast packet themselves.

### 9.5.2.8 Properties

This QosPolicy cannot be modified after the `DomainParticipant` is created.

It can be set differently on the publishing and subscribing sides.

### 9.5.2.9 Related QosPolicies

- 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) below
- 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665

### 9.5.2.10 Applicable Entities

- 9.3 DomainParticipants on page 609

### 9.5.2.11 System Resource Considerations

For every entry in the `initial_peers` list, Connext DDS will periodically send a discovery packet to see if that participant exists. If the list has many potential participants that are never started, then CPU and network bandwidth may be wasted in sending out packets that will never be received.

### 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension)

The DISCOVERY_CONFIG QosPolicy is used to tune the discovery process. It controls how often to send discovery packets, how to determine when participants are alive or dead, and resources used by the discovery mechanism.

The amount of network traffic required by the discovery process can vary widely based on how your application has chosen to configure the middleware's network addressing (unicast vs. multicast, multicast TTL, etc.), the size of the system, whether all applications are started at the same time or whether start times are staggered, and other factors. Your application can use this policy to make trade-offs between discovery completion time and network bandwidth utilization. In addition, you can introduce random back-off periods into the discovery process to decrease the probability of network contention when many applications start simultaneously.

This QosPolicy includes the members in Table 9.11 DDS_DiscoveryConfigQosPolicy. Many of these members are described in Discovery (Chapter 15 on page 764). For defaults and valid ranges, please refer to the API Reference HTML documentation.
Table 9.11 DDS_DiscoveryConfigQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Duration_t</td>
<td>participant_liveliness_lease_duration</td>
<td>The time period after which other DomainParticipants can consider this one dead if they do not receive a liveliness packet from this DomainParticipant.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>participant_liveliness_assert_period</td>
<td>The period of time at which this DomainParticipant will send out packets asserting that it is alive.</td>
</tr>
<tr>
<td>DDS_RemoteParticipantPurgeKind</td>
<td>remote_participant_purge_kind</td>
<td>Controls the DomainParticipant’s behavior for purging records of remote participants (and their contained entities) with which discovery communication has been lost. See 9.5.3.2 Controlling Purging of Remote Participants on page 652.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>max_liveliness_loss_detection_period</td>
<td>The maximum amount of time between when a remote entity stops maintaining its liveliness and when the matched local entity realizes that fact.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_participant_announcements</td>
<td>Sets how many initial liveliness announcements the DomainParticipant will send when it is first enabled, or after discovering a new remote participant.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>min_initial_participant_announcement_period</td>
<td>Sets the minimum and maximum times between liveliness announcements.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>When a participant is first enabled, or after discovering a new remote participant, Connext DDS sends initial_participant_announcements number of discovery messages. These messages are sent with a sleep period between them that is a random duration between min_initial_participant_announcement_period and max_initial_participant_announcement_period.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>max_initial_participant_announcement_period</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.12 DDS_BuiltinTopicReaderResourceLimits_t | participant_reader_resource_limits | Configures the resource for the built-in DataReaders used to access discovery information; see 9.5.3.1 Resource Limits for Built-In Topic DataReaders on page 650 and Built-In Topics (Chapter 18 on page 831). |

Table 8.19 DDS_RtpsReliableReaderProtocol_t | publication_reader | Configures the RTPS reliable protocol parameters for a built-in publication reader.                                                                                                                         |

Table 9.12 DDS_BuiltinTopicReaderResourceLimits_t | publication_reader_resource_limits | Configures the resource for the built-in DataReaders used to access discovery information; see 9.5.3.1 Resource Limits for Built-In Topic DataReaders on page 650 and Built-In Topics (Chapter 18 on page 831). |

Table 8.19 DDS_RtpsReliableReaderProtocol_t | subscription_reader | Configures the RTPS reliable protocol parameters for a built-in subscription reader. Built-in subscription readers receive discovery information reliably from DomainParticipants that were dynamically discovered (see Discovery (Chapter 15 on page 764)). |

Table 9.12 DDS_BuiltinTopicReaderResourceLimits_t | subscription_reader_resource_limits | Configures the resource for the built-in DataReaders used to access discovery information; see 9.5.3.1 Resource Limits for Built-In Topic DataReaders on page 650 and Built-In Topics (Chapter 18 on page 831). |
# Table 9.11 DDS_DiscoveryConfigQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 7.45 DDS_RtpsReliableWriterProtocol</td>
<td>publication_writer</td>
<td>Configures the RTPS reliable protocol parameters for the writer side of a reliable connection. Built-in DataWriters send reliable discovery information to DomainParticipants that were dynamically discovered (see Discovery (Chapter 15 on page 764)).</td>
</tr>
<tr>
<td>Table 7.80 DDS_Writer-DataLifecycleQosPolicy</td>
<td>publication_writer_data_lifecycle</td>
<td>Configures writer data-lifecycle settings for a built-in publication writer. (DDS_WriterDataLifecycleQosPolicy::autodispose_unregistered_instances will always be TRUE.)</td>
</tr>
<tr>
<td>Table 7.45 DDS_RtpsReliableWriterProtocol</td>
<td>subscription_writer</td>
<td>Configures the RTPS reliable protocol parameters for the writer side of a reliable connection. Built-in DataWriters send reliable discovery information to DomainParticipants that were dynamically discovered (see Discovery (Chapter 15 on page 764)).</td>
</tr>
<tr>
<td>Table 7.80 DDS_Writer-DataLifecycleQosPolicy</td>
<td>subscription_writer_data_lifecycle</td>
<td>Configures writer data-lifecycle settings for a built-in subscription writer. (DDS_WriterDataLifecycleQosPolicy::autodispose_unregistered_instances will always be TRUE.)</td>
</tr>
<tr>
<td>DDS_DiscoveryConfigBuiltInPluginKindMask</td>
<td>builtin_discovery_plugins</td>
<td>The bit mask of available kinds for selecting builtin discovery plugins:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• (default) DDS_DISCOVERYCONFIG_BUILTIN SDP: Enables the builtin Simple Discovery Protocol, which consists of both the 15.1.1 Simple Participant Discovery on page 765 (SPDP) and the 15.1.2 Simple Endpoint Discovery on page 766 (SEDP).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_DISCOVERYCONFIG_BUILTIN_SPDP: Enables only the 15.1.1 Simple Participant Discovery on page 765, which means that you intend to use a non-builtin alternative for endpoint discovery, such as Limited Bandwidth Endpoint Discovery (LBED).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_DISCOVERYCONFIG_BUILTIN_SEDP: Enables only the 15.1.2 Simple Endpoint Discovery on page 766, which means that you intend to use a non-builtin alternative for participant discovery, such as Limited Bandwidth Participant Discovery (LBPD).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_DISCOVERYCONFIG_BUILTIN_PLUGIN_MASK_NONE: No builtin discovery is used. This setting should be used if you are replacing both endpoint and participant discovery algorithms with non-builtin alternatives. For example, MASK_NONE can be used when LBED and LBPD are both (simultaneously) enabled via the Limited Bandwidth Plugins.</td>
</tr>
<tr>
<td>DDS_ReliabilityQosPolicyKind</td>
<td>participant_message_reader_reliability_kind</td>
<td>Reliability kind configuration setting for a built-in participant message reader (default: best-effort). See Table 7.67 DDS_ReliabilityQosPolicy</td>
</tr>
<tr>
<td>Table 8.19 DDS_RtpsReliableReaderProtocol</td>
<td>participant_message_reader</td>
<td>RTPS protocol-related configuration settings for a built-in participant message reader.</td>
</tr>
</tbody>
</table>
### Table 9.11 DDS_DiscoveryConfigQosPolicy

<table>
<thead>
<tr>
<th>Type<img src="field_type.png" alt="" /></th>
<th>Field Name<img src="field_name.png" alt="" /></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 7.45 DDS_RtpsReliableWriterProtocol_t</td>
<td>participant_message_writer</td>
<td>RTPS protocol-related configuration settings for a built-in participant message writer.</td>
</tr>
<tr>
<td>Table 7.66 DDS_PublishModeQosPolicy</td>
<td>publication_writer_publish_mode</td>
<td>Determines whether the Discovery built-in publication DataWriter publishes data synchronously or asynchronously and how.</td>
</tr>
<tr>
<td>Table 7.66 DDS_PublishModeQosPolicy</td>
<td>subscription_writer_publish_mode</td>
<td>Determines whether the Discovery built-in subscription DataWriter publishes data synchronously or asynchronously and how.</td>
</tr>
<tr>
<td>Table 7.19 DDS_AsynchronousPublisherQosPolicy</td>
<td>asynchronous_publisher</td>
<td>Asynchronous publishing settings for the Discovery Publisher and all entities that are created by it.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>default_domain_announcement_period</td>
<td>The period at which a participant will announce itself to the default DDS domain 0 using the default UDPv4 multicast group address for discovery traffic on that DDS domain. For DDS domain 0, the default discovery multicast address is 239.255.0.1:7400. To disable announcement to the default DDS domain, set this to DURATION_INFINITE. When this period is set to a value other than DURATION_INFINITE and <strong>ignore_default_domain_announcements</strong> (see below) is FALSE, you can get information about participants running in different DDS domains by creating a participant in DDS domain 0 and implementing the <strong>on_data_available</strong> callback (see 8.3.7.1 DATAAVAILABLE Status on page 517) in the ParticipantBuiltinTopicData built-in DataReader’s listener (see 18.2 Built-in DataReaders on page 832). You can learn the domain ID associated with a participant by looking at the domain_id on page 833 in the ParticipantBuiltinTopicData.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>ignore_default_domain_announcements</td>
<td>When TRUE, ignores the announcements received by a participant on the default DDS domain 0 corresponding to participants running on domains IDs other than 0. This setting only applies to participants running on the default DDS domain 0 and using the default port mapping. When TRUE, a participant running on the default DDS domain 0 will ignore announcements from participants running on different DDS domain IDs. When FALSE, a participant running on the default DDS domain 0 will provide announcements from participants running on different DDS domain IDs to the application via the ParticipantBuiltinTopicData built-in DataReader (see 18.2 Built-in DataReaders on page 832).</td>
</tr>
<tr>
<td>Table 7.45 DDS_RtpsReliableWriterProtocol_t</td>
<td>service_request_writer</td>
<td>RTPS protocol-related configuration settings for the built-in service request writer.</td>
</tr>
<tr>
<td>Table 7.80 DDS_Writer_DataLifecycleQosPolicy</td>
<td>service_request_writer_data_lifecycle</td>
<td>Configures writer data-lifecycle settings for the built-in service request writer.</td>
</tr>
<tr>
<td>Table 7.66 DDS_PublishModeQosPolicy</td>
<td>service_request_writer_publish_mode</td>
<td>Determines whether the Discovery built-in service request DataWriter publishes data synchronously or asynchronously and how.</td>
</tr>
</tbody>
</table>
### Table 9.11 DDS_DiscoveryConfigQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Table 8.18 DDS_DataReaderProtocolQosPolicy</strong></td>
<td>service_request_reader</td>
<td>RTPS protocol-related configuration settings for the built-in service request reader.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>locator_reachability_assert_period</td>
<td>Configures the period at which this DomainParticipant will ping all the locators that it has discovered from other DomainParticipants. This period should be strictly less than locator_reachability_lease_duration below. If locator_reachability_lease_duration below is INFINITE, this parameter is ignored. The DomainParticipant will not assert remote locators.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>locator_reachability_lease_duration</td>
<td>For the purpose of this explanation, we use 'local' to refer to the DomainParticipant in which we configure locator_reachability_lease_duration above and 'remote' to refer to the other DomainParticipants communicating with the local DomainParticipant. This setting configures a timeout announced to the remote DomainParticipants. This timeout is used by the remote DomainParticipants as the maximum period by which a remote locator must be asserted by the local DomainParticipant (through a REACHABILITY PING message) before considering this locator as &quot;unreachable&quot; from the local DomainParticipant. When a remote DomainParticipant detects that one of its locators is not reachable from the local DomainParticipant, it will notify the local DomainParticipant of this event. From that moment on, and until notified otherwise, the local DomainParticipant will not send RTPS messages to remote DomainParticipants using this locator. If this value is set to INFINITE, the local DomainParticipant will send RTPS messages to a remote DomainParticipant on the locators announced by the remote DomainParticipant, regardless of whether or not the remote DomainParticipant can be reached using these locators.</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>locator_reachability_change_detection_period</td>
<td>Determines the maximum period at which this DomainParticipant will check to see if its locators are reachable from other DomainParticipants according to the other DomainParticipants' locator_reachability_lease_duration above. If locator_reachability_lease_duration above is INFINITE, this parameter is ignored. The DomainParticipant will not schedule an event to see if its locators are reachable from other DomainParticipants.</td>
</tr>
<tr>
<td><strong>Table 7.45 DDS_RtpsReliableWriterProtocol_t</strong></td>
<td>secure_volatile_writer</td>
<td>RTPS protocol-related configuration settings for the built-in Key Exchange writer.</td>
</tr>
<tr>
<td><strong>Table 7.66 DDS_PublishModeQosPolicy</strong></td>
<td>secure_volatile_writer_publish_mode</td>
<td>Publish mode policy for the builtin secure volatile writer. Determines whether the builtin secure volatile DataWriter publishes data synchronously or asynchronously and how.</td>
</tr>
</tbody>
</table>
### Table 9.11 DDS_DiscoveryConfigQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 8.19 DDS_RtpeReliableReaderProtocol_t</td>
<td>secure_volatile_reader</td>
<td>RTPS protocol-related configuration settings for the built in Key Exchange reader.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>endpoint_type_object_lb_serialization_threshold</td>
<td>Minimum size (in bytes) of the serialized TypeObject that will trigger the serialization of a TypeObjectLb instead of the regular TypeObject. TypeObjectLb is a compressed version of the serialized TypeObject. This compressed version reduces the size needed to propagate a TypeObject as part of Simple Endpoint Discovery. For example, setting this policy to 1000 will trigger the serialization of the TypeObjectLb for TypeObjects whose serialized size is greater than 1000 Bytes. Range: [-1, 2147483647]. The sentinel value -1 disables TypeObject compression (by never sending TypeObjectLb). Any non-valid values will behave as 0. Default: 0 (TypeObjectLb is enabled by default)</td>
</tr>
<tr>
<td>DDS_Duration_t</td>
<td>dns_tracker.polling_period</td>
<td>Configures the frequency used by the DNS Tracker thread to query the DNS service. If this parameter is set to INFINITE, the DNS tracker is disabled and changes in hostnames will not be tracked. See 17.3 Using DNS Tracker to Keep Peer List Updated on page 830 for more information.</td>
</tr>
</tbody>
</table>

A DomainParticipant needs to send a message periodically to other DomainParticipants to let the other participants know that it is still alive. These liveliness messages are sent to all peers in the peer list that was initialized by the initial_peers parameter of the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640. Peer participants on the peer list may or may not be alive themselves. The peer DomainParticipants that already know about this DomainParticipant will use the participant_liveliness_lease_duration provided by this participant to declare the participant dead, if they have not received a liveliness message for the specified time.

The participant_liveliness_assert_period is the periodic rate at which this DomainParticipant will be sending liveliness messages. Since these liveliness messages are not sent reliably and can get dropped by the transport, it is important to set:

\[
\text{participant_liveliness_assert_period} < \text{participant_liveliness_lease_duration}/N
\]

where N is the number of liveliness messages that other DomainParticipants must miss before they decide that this DomainParticipant is dead.

DomainParticipants that receive a liveliness message from a participant that they did not know about previously will have “discovered” the participant. When one DomainParticipant discovers another, the discoverer will immediately send its own liveliness packets back. initial_participant_announcements controls how many of these initial liveliness messages are sent, and max_initial_participant_announcement_period controls the time period in between each message.
After the initial set of liveness messages are sent (when the DomainParticipant is first enabled or discovers a new remote participant), the DomainParticipant will return to sending liveness packets to all peers in its peer list at the rate governed by `participant_liveliness_assert_period`.

For more information on the discovery process, see Discovery (Chapter 15 on page 764).

### 9.5.3.1 Resource Limits for Builtin-Topic DataReaders

The DDS_BuiltinTopicReaderResourceLimits_t structure is shown in Table 9.12 DDS_BuiltinTopicReaderResourceLimits_t. This structure contains several fields that are used to configure the resource limits of the builtin-topic DataReaders used to receive discovery meta-traffic from other DomainParticipants.

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td><code>initial_samples</code></td>
<td>Initial number of meta-traffic DDS data samples that can be stored by a builtin-topic DataReader.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td><code>max_samples</code></td>
<td>Maximum number of meta-traffic DDS data samples that can be stored by a builtin-topic DataReader.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td><code>initial_infos</code></td>
<td>Initial number of DDS_SampleInfo structures allocated for the builtin-topic DataReader.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td><code>max_infos</code></td>
<td>Maximum number of DDS_SampleInfo structures that can be allocated for the builtin-topic DataReader.</td>
</tr>
<tr>
<td></td>
<td><code>max_infos</code> must be &gt;= <code>max_samples</code></td>
<td></td>
</tr>
<tr>
<td>DDS_Long</td>
<td><code>initial_outstanding_reads</code></td>
<td>Initial number of times in which memory can be concurrently loaned via read/take calls on the builtin-topic DataReader without being returned with <code>return_loan()</code>.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td><code>max_outstanding_reads</code></td>
<td>Maximum number of times in which memory can be concurrently loaned via read/take calls on the builtin-topic DataReader without being returned with <code>return_loan()</code>.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td><code>max_samples_per_read</code></td>
<td>Maximum number of DDS samples that can be read/taken on a same built-in topic DataReader.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td><code>disable_fragmentation_support</code></td>
<td>Determines whether the builtin-topic DataReader can receive fragmented DDS samples. When fragmentation support is not needed, disabling fragmentation support will save some memory resources.</td>
</tr>
</tbody>
</table>
Table 9.12 DDS_BuiltinTopicReaderResourceLimits_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>max_fragmented_samples</td>
<td>The maximum number of DDS samples for which the DataReader may store fragments at a given point in time. At any given time, a DataReader may store fragments for up to max_fragmented_samples DDS samples while waiting for the remaining fragments. These DDS samples need not have consecutive sequence numbers and may have been sent by different DataWriters. Once all fragments of a DDS sample have been received, the DDS sample is treated as a regular DDS sample and becomes subject to standard QoS settings, such as max_samples. Connext DDS will drop fragments if the max_fragmented_samples limit has been reached. For best-effort communication, Connext DDS will accept a fragment for a new DDS sample, but drop the oldest fragmented DDS sample from the same remote writer. For reliable communication, Connext DDS will drop fragments for any new DDS samples until all fragments for at least one older DDS sample from that writer have been received. Only applies if disable_fragmentation_support is FALSE.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_fragmented_samples</td>
<td>The initial number of DDS samples for which a builtin-topic DataReader may store fragments. Only applies if disable_fragmentation_support on the previous page is FALSE.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_fragmented_samples_per_remote_writer</td>
<td>The maximum number of DDS samples per remote writer for which a builtin-topic DataReader may store fragments. Logical limit so a single remote writer cannot consume all available resources. Only applies if disable_fragmentation_support on the previous page is FALSE.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_fragmented_per_sample</td>
<td>Maximum number of fragments for a single DDS sample. Only applies if disable_fragmentation_support on the previous page is FALSE.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>dynamically_allocate_fragmented_samples</td>
<td>By default, the middleware does not allocate memory upfront, but instead allocates memory from the heap upon receiving the first fragment of a new sample. The amount of memory allocated equals the amount of memory needed to store all fragments in the sample. Once all fragments of a sample have been received, the sample is deserialized and stored in the regular receive queue. At that time, the dynamically allocated memory is freed again. This QoS setting is useful for large, but variable-sized data types where up-front memory allocation for multiple samples based on the maximum possible sample size may be expensive. The main disadvantage of not pre-allocating memory is that one can no longer guarantee the middleware will have sufficient resources at run-time. If dynamically_allocate_fragmented_samples is FALSE, the middleware will allocate memory up-front for storing fragments for up to initial_fragmented_samples samples. This memory may grow up to max_fragmented_samples if needed. Only applies if disable_fragmentation_support on the previous page is FALSE.</td>
</tr>
</tbody>
</table>

There are builtin-topics for exchanging data about DomainParticipants, for publications (Publisher/DataWriter combination) and for subscriptions (Subscriber/DataReader combination). The DataReaders for the publication and subscription builtin-topics are reliable. The DataReader for the participant builtin-topic is best effort.

You can set listeners on these DataReaders that are created automatically when a DomainParticipant is created. With these listeners, your code can be notified when remote DomainParticipants, Publishers/DataWriters, and Subscriber/DataReaders are discovered. You can always check the receive
queues of those DataReaders for the same information about discovered entities at any time. Please see Built-In Topics (Chapter 18 on page 831) for more details.

The initial_samples and max_samples, and related initial_infos and max_infos, fields size the amount of declaration messages can be stored in each builtin-topic DataReader.

9.5.3.2 Controlling Purging of Remote Participants

When discovery communication with a remote participant has been lost, the local participant must make a decision about whether to continue attempting to communicate with that participant and its contained entities. The remote_participant_purge_kind is used to select the desired behavior.

This does not pertain to the situation in which a remote participant has been gracefully deleted and notification of that deletion has been successfully received by its peers. In that case, the local participant will immediately stop attempting to communicate with those entities and will remove the associated remote entity records from its internal database.

The remote_participant_purge_kind can be set to the following values:

**DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE**

This value causes Connext DDS to keep the state of a remote participant and its contained entities for as long as the participant maintains its liveliness contract (as specified by its participant_liveliness_lease_duration in the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644).

A participant will maintain its own liveliness to any remote participant via inter-participant liveliness traffic (see 7.5.15 LIVELINESS QosPolicy on page 427).

The default Simple Discovery Protocol described in Discovery (Chapter 15 on page 764) automatically maintains this liveliness, whereas other discovery mechanisms may or may not.

**DDS_NO_REMOTE_PARTICIPANT_PURGE**

With this value, Connext DDS will never purge the records of a remote participant with which discovery communication has been lost.

- If the remote participant is later rediscovered, the records that remain in the database will be re-used.
- If the remote participant is not rediscovered, the records will continue to take up space in the database for as long as the local participant remains in existence.

In most cases, you will not need to change this value from its default, DDS_LIVELINESS_BASED_REMOTE_PARTICIPANT_PURGE.

However, DDS_NO_REMOTE_PARTICIPANT_PURGE may be a good choice if the following conditions apply:
Discovery communication with a remote participant may be lost while data communication remains intact. This will not be the typical case if discovery takes place over the Simple Discovery Protocol.

Extensive and prolonged lack of discovery communication between participants is not expected to be common, either because loss of the participant will be rare, or because participants may be lost sporadically but will typically return again.

Maintaining inter-participant liveliness is problematic, perhaps because a participant has no writers with the appropriate 7.5.15 LIVELINESS QosPolicy on page 427 kind.

9.5.3.3 Controlling the Reliable Protocol Used by Built-in-Topic DataWriters/DataReaders

The connection between the DataWriters and DataReaders for the publication and subscription built-in-topics are reliable. The publication_writer, subscription_writer, publication_reader, and subscription_reader parameters of the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644 configure the reliable messaging protocol used by Connext DDS for those topics. Connext DDS’s reliable messaging protocol is discussed in Reliable Communications (Chapter 11 on page 688).

See also:

- 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389
- 8.6.1 DATAREADER_PROTOCOL QosPolicy (DDS Extension) on page 570.

9.5.3.4 Example

Users will be most interested in setting the participant_liveliness_lease_duration and participant_liveliness_assert_period values for their DomainParticipants. Basically, the lease duration governs how fast an application realizes another application dies unexpectedly. The shorter the periods, the quicker a DomainParticipant can determine that a remote participant is dead and act accordingly by declaring all of the remote DataWriters and DataReaders of that participant dead as well.

However, you should realize that the shorter the period the more liveliness packets will sent by the DomainParticipant. How many packets is also determined by the number of peers in the peer list of the participant—whether or not the peers on the list are actually alive.

9.5.3.5 Properties

This QosPolicy cannot be modified after the DomainParticipant is created.

It can be set differently on the publishing and subscribing sides.

9.5.3.6 Related QosPolicies

- 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640
- 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) below
9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670
7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389
8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570
8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576

9.5.3.7 Applicable DDS Entities

- 9.3 DomainParticipants on page 609

9.5.3.8 System Resource Considerations

Setting smaller values for time periods can increase the CPU and network bandwidth usage. Setting larger values for maximum limits can increase the maximum memory that Connext DDS may allocate for a DomainParticipant while increasing the initial values will increase the initial memory allocated for a DomainParticipant.

9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension)

The DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy includes various settings that configure how DomainParticipants allocate and use physical memory for internal resources, including the maximum sizes of various properties.

This QosPolicy sets maximum size limits on variable-length parameters used by the participant and its contained Entities. It also controls the initial and maximum sizes of data structures used by the participant to store information about locally-created and remotely-discovered entities (such as DataWriters/DataReaders), as well as parameters used by the internal database to size the hash tables used by the data structures.

By default, a DomainParticipant is allowed to dynamically allocate memory as needed as users create local Entities such as DataWriters and DataReaders or as the participant discovers new applications to store their information. By setting fixed values for the maximum parameters in this QosPolicy, you can bound the memory that can be allocated by a DomainParticipant. In addition, by setting the initial values to the maximum values, you can reduce the amount of memory allocated by DomainParticipants after the initialization period. Notice that memory can still be allocated dynamically after the initialization period. For example, when a new local DataWriter or DataReader is created, the initial memory required for its queue is allocated dynamically.

The maximum sizes of several variable-length parameters—such as the number of partitions that can be stored in the 7.4.5 PARTITION QosPolicy on page 356, the maximum length of data store in the 7.5.30 USER_DATA QosPolicy on page 465 and 7.4.4 GROUP_DATA QosPolicy on page 353, and many others—can be changed from their defaults using this QoS. However, it is important that all DomainParticipants that need to communicate with each other use the same set of maximum values. Otherwise, when these parameters are propagated from one DomainParticipant to another, a DomainParticipant with a smaller maximum length may reject the parameter resulting in an error.
This QosPolicy includes the members in Table 9.13 DDS_DomainParticipantResourceLimitsQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

### Table 9.13 DDS_DomainParticipantResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Allocation-Settings_t (see description column)</td>
<td>local_writer_allocation</td>
<td>Each allocation structure configures how many objects of each type, <code>&lt;object&gt;_&lt;allocation&gt;</code>, will be allocated by the DomainParticipant. See 9.5.4.1 Configuring Resource Limits for Asynchronous DataWriters on page 660.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDS AllocationSettings_t</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDS_Long initial_count;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDS_Long max_count;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DDS_Long incremental_count;</td>
</tr>
<tr>
<td></td>
<td>remote_writer_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>local_reader_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>local_publisher_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>local_subscriber_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>local_topic_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>remote_writer_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>remote_reader_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>remote_participant_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>matching_writer_reader_pair_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>matching_reader_writer_pair_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>ignored_entity_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>content_filtered_topic_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>content_filter_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>read_condition_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>query_condition_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>outstanding_asynchronous_sample_allocation</td>
<td>See above row</td>
</tr>
<tr>
<td></td>
<td>flow_controller_allocation</td>
<td>See above row</td>
</tr>
</tbody>
</table>
### Table 9.13 DDS_DomainParticipantResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| DDS_DomainParticipant         | ResourceLimitsIgnoredEntityReplacementKind      | Sets the kinds of entities allowed to be replaced when a DomainParticipant reaches ign
el_entity_allocation.max_count. See 18.4.4 Resource Limits Considerations for Ignored Entities on page 845.                                          |
| DDS_Long                      | local_writer_hash_buckets                       | Used to configure the hash tables used for database searches. If these numbers are too large then memory is wasted. If these number are too small, searching for an object will be less efficient.        |
| DDS_Long                      | local_reader_hash_buckets                       | See above row                                                                                                                                                                                                |
| DDS_Long                      | local_publisher_hash_buckets                    | See above row                                                                                                                                                                                                |
| DDS_Long                      | local_subscriber_hash_buckets                   | See above row                                                                                                                                                                                                |
| DDS_Long                      | local_topic_hash_buckets                        | See above row                                                                                                                                                                                                |
| DDS_Long                      | remote_writer_hash_buckets                      | See above row                                                                                                                                                                                                |
| DDS_Long                      | remote_reader_hash_buckets                      | See above row                                                                                                                                                                                                |
| DDS_Long                      | remote_participant_hash_buckets                 | See above row                                                                                                                                                                                                |
| DDS_Long                      | matching_writer_reader_pair_hash_buckets        | See above row                                                                                                                                                                                                |
| DDS_Long                      | matching_reader_writer_pair_hash_buckets        | See above row                                                                                                                                                                                                |
| DDS_Long                      | ignored_entity_hash_buckets                     | See above row                                                                                                                                                                                                |
| DDS_Long                      | content_filtered_topic_hash_buckets             | See above row                                                                                                                                                                                                |
| DDS_Long                      | content_filter_hash_buckets                     | See above row                                                                                                                                                                                                |
| DDS_Long                      | flow_controller_hash_buckets                    | See above row                                                                                                                                                                                                |
| DDS_Long                      | max_gather_destinations                         | Configures the maximum number of destinations that a message can be addressed in a single network send operation. Can improve efficiency if the underlying transport support can send to multiple destinations. |
| DDS_Long                      | participant_user_data_max_length                | Controls the maximum lengths of 7.5.30 USER_DATA QosPolicy on page 465, 5.2.1 TOPIC_DATA QosPolicy on page 224 and 7.4.4 GROUP_DATA QosPolicy on page 353 for different entities. Must be configured to be the same values on all DomainParticipants in the same DDS domain. |
| DDS_Long                      | topic_data_max_length                           | See above row                                                                                                                                                                                                |
| DDS_Long                      | publisher_group_data_max_length                  | See above row                                                                                                                                                                                                |
### Table 9.13 DDS_DomainParticipantResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>subscriber_group_data_max_length</td>
<td>See above row</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>writer_user_data_max_length</td>
<td>See above row</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>reader_user_data_max_length</td>
<td>See above row</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_partitions</td>
<td>Controls the maximum number of partitions that can be assigned to a Publisher or Subscriber with the 7.4.5 PARTITION QosPolicy on page 356. Must be configured to be the same value on all DomainParticipants in the same DDS domain.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_partition_cumulative_characters</td>
<td>Controls the maximum number of combined characters among all partition names in the 7.4.5 PARTITION QosPolicy on page 356. Must be configured to be the same value on all DomainParticipants in the same DDS domain.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>type_code_max_serialized_length</td>
<td>Maximum size of serialized string for type code. If your data type has an especially complex type code, you may need to increase this value. See 3.7 Using Generated Types without Connext DDS (Standalone) on page 152. Note: TypeObject is now the standard method of exchanging type information in Connext DDS, so <code>type_code_max_serialized_length</code> defaults to 0 bytes. It is recommended to use <code>type_object_max_serialized_length</code> to configure the maximum serialized size for the TypeObject describing the type.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>type_object_max_serialized_length</td>
<td>Maximum length, in bytes, that the buffer to serialize TypeObject can consume. This parameter limits the size of the TypeObject that a DomainParticipant is able to propagate. Since TypeObjects contain all of the information of a data structure, including the strings that define the names of the members of a structure, complex data-structures can result in TypeObjects larger than the default maximum. This field allows you to specify a larger value. Cannot be unlimited.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>type_object_max_deserialized_length</td>
<td>Maximum number of bytes that a deserialized TypeObject can consume. This parameter limits the size of the TypeObject that a DomainParticipant is able to store.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>serialized_type_object_dynamic_allocation_threshold</td>
<td>Threshold, in bytes, for dynamic memory allocation for the serialized typeObject. Above this threshold, the memory for a TypeObject is allocated dynamically. Below it, the memory is obtained from a pool of fixed-size buffers. If <code>type_object_max_serialized_length</code> is not LENGTH_UNLIMITED and is smaller than <code>serialized_type_object_dynamic_allocation_threshold</code>, then <code>serialized_type_object_dynamic_allocation_threshold</code> will be adjusted to <code>type_object_max_serialized_length</code> and a warning will be logged. By default, <code>serialized_type_object_dynamic_allocation_threshold</code> is the same value as <code>type_object_max_serialized_length</code>, 8192. This means that the TypeObject memory is obtained from a pool of fixed-size buffers.</td>
</tr>
</tbody>
</table>
### Table 9.13 DDS_DomainParticipantResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>deserialized_type_object_dynamic_allocation_threshold</td>
<td>Threshold, in bytes, for dynamic memory allocation for the deserialized TypeObject. Above this threshold, the memory for a TypeObject is allocated dynamically. Below it, the memory is obtained from a pool of fixed-size buffers. The size of the buffers is equal to this threshold. If <code>type_object_max_deserialized_length</code> is not <code>LENGTH_UNLIMITED</code> and is smaller than <code>deserialized_type_object_dynamic_allocation_threshold</code>, then <code>deserialized_type_object_dynamic_allocation_threshold</code> will be adjusted to <code>type_object_max_deserialized_length</code> and a warning will be logged.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>contentfilter_property_max_length</td>
<td>Maximum length of all data related to 5.4 ContentFilteredTopics on page 227.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>channel_seq_max_length</td>
<td>Maximum number of channels that can be specified in a DataWriter’s 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>channel_filter_expression_max_length</td>
<td>Maximum length of a channel filter_expression in a DataWriter’s 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>participant_property_list_max_length</td>
<td>Maximum number of properties ((name, value) pairs) that can be stored in the DomainParticipant’s 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>participant_property_string_max_length</td>
<td>Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a DomainParticipant’s Property QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>writer_property_list_max_length</td>
<td>Maximum number of properties ((name, value) pairs) that can be stored in a DataWriter’s Property QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>writer_property_string_max_length</td>
<td>Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a DataWriter’s Property QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>reader_property_list_max_length</td>
<td>Maximum number of properties ((name, value) pairs) that can be stored in a DataReader’s Property QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>reader_property_string_max_length</td>
<td>Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a DataReader’s Property QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_endpoint_groups</td>
<td>Maximum number of endpoint groups allowed in an 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_endpoint_group_cumulative_characters</td>
<td>Maximum number of combined <code>role_name</code> characters allowed in all endpoint groups in an 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370. The maximum number of combined characters should account for a terminating NULL (”) character for each <code>role_name</code> string.</td>
</tr>
</tbody>
</table>
9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension)

### Table 9.13 DDS_DomainParticipantResourceLimitsQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>transport_info_list_max_length</td>
<td>When sending DomainParticipant discovery information, this value defines the maximum number of transports whose properties will be announced to other DomainParticipants. If a DomainParticipant has three transports installed and this value is two, the DomainParticipant will only announce information about the first two transports. When receiving DomainParticipant information, this value defines the maximum size of the list containing information about the transports installed in a remote DomainParticipant. The information about the transports installed in a DomainParticipant is made available to remote DomainParticipants through the sequence field transport_info in the Participant Built-in Topic's Data (see Table 18.1 Participant Built-in Topic's Data Type (DDS_ParticipantBuiltInTopicData)). Setting this value to 0 disables the capability of Connext DDS to detect and report transport misconfigurations. However, it does not affect the capability of reaching a given DomainParticipant in all transports available on that DomainParticipant.</td>
</tr>
<tr>
<td>DDS_AllocationSettings_t</td>
<td>remote_topic_query_allocation</td>
<td>Allocation settings applied to remote TopicQueries. These settings are applied to the allocation of information about TopicQueries created by other participants and discovered by this participant. When the participant receives a new topic query that would make the current count go above max_count, it is not processed until the current count drops (i.e. another topic query is canceled). The topic query stays in the Built-in ServiceRequest DataReader queue until it can be processed or it is canceled.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>remote_topic_query_hash_buckets</td>
<td>Number of hash buckets for remote TopicQueries.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>writer_data_tag_list_max_length</td>
<td>Maximum number of data tags (name, value) pairs that can be stored in a DataWriter's DataTag QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>writer_data_tag_string_max_length</td>
<td>Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a DataWriter's DataTag QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>reader_data_tag_list_max_length</td>
<td>Maximum number of data tags (name, value) pairs that can be stored in a DataReader's DataTag QosPolicy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>reader_data_tag_string_max_length</td>
<td>Maximum cumulative length (in bytes, including the null terminating characters) of all the (name, value) pairs in a DataReader's DataTag QosPolicy.</td>
</tr>
<tr>
<td>DDS_UnsignedLong</td>
<td>shmem_ref_transfer_mode_max_segments</td>
<td>Sets the maximum number of shared memory segments that can be created by all Data Writers belonging to this participant if you are using Zero Copy transfer over shared memory. See 23.1.5 Zero Copy Transfer Over Shared Memory on page 962.</td>
</tr>
</tbody>
</table>

Most of the parameters for this QosPolicy are described in the Description column of the table. However, you may need to refer to the sections listed in the column to fully understand the context in which the parameter is used.

An important parameter in this QosPolicy that is often changed by users is the **type_object_max_serialized_length**. This parameter limits the size of the TypeObject that a DomainParticipant is able to store and propagate for user data types. TypeObjects are the wire representation for a type code. Type codes can be used by external applications to understand user data types without having the data type predefined in
9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension)

compiled form. However, since type codes contain all of the information of a data structure, including the strings that define the names of the members of a structure, complex data structures can result in TypeOb-jects larger than the default maximum of 8192 bytes. Thus it is common for users to set this parameter to a larger value. However, as with all parameters in this QosPolicy defining maximum sizes for variable-length elements, all DomainParticipants should set the same value for type_object_max Serialized_length.

The <object type> hash_buckets configure the hash-table data structure that is used to efficiently search the database. The optimal number of buckets depend on the actual number of objects that will be stored in the hash table. So if you know how many Datawriters will be created in a DomainParticipant, you may change the value of local_writer_hash_buckets to balance memory usage against search efficiency. A smaller value will use up less memory, but a larger value will make database lookups for the object more effi-
cient.

If you modify any of the <entity type> data_max_length, max_partitions, or max_partition_cumulative_characters parameters, then you must make sure that they are modified to be the same value for all DomainParticipants in the same DDS domain for all applications. If they are different and an application sends data that is larger than another application is configure to hold, then the two Entities, whether a matching DataWriter!DataReader pair or even two DomainParticipants will fail to connect.

9.5.4.1 Configuring Resource Limits for Asynchronous DataWriters

When using an asynchronous Publisher, if a call to write() is blocked due to a resource limit, the block will last until the timeout period expires, which will prevent others from freeing the resource. To avoid this situation, make sure that the DomainParticipant’s resource_limits.outstanding_asynchronous_sample_allocation is always greater than the sum of all asynchronous DataWriters’ resource_limits.max_samples (see 7.5.22 RESOURCE_LIMITS QosPolicy on page 451).

9.5.4.2 Configuring Memory Allocation

The <object type>_allocation configures the number of <object type>’s that can be stored in the internal Connext DDS database. For example, local_writer_allocation configures how many local DataWriters can be created for the DomainParticipant.

The DDS_AllocationSettings_t structure sets the initial and maximum number of each object type that can be stored. The initial_count will determine how many objects are initially allocated, and max_count will determine the maximum amount of objects that Connext DDS is allowed to allocate. The incremental_count is used to allocate more objects in chunks when the number of objects created exceed the initial_count. You can used fixed-size increments or -1 to double the amount of extra memory allocated each time memory is needed.

Notice that the memory pre-allocated for an object using the DDS_AllocationSettings_t structure is not the full memory that will be required by the object during its lifecycle. Memory can still be allocated dynamically when the object is actually used. For example, when a new local DataWriter or DataReader is created, the memory required for its queue is allocated from the heap dynamically at the moment of creation,
independently of the DDS_AllocationSettings_t value. The memory pre-allocated for the object by using the DDS_AllocationSettings_t structure only accounts for the memory required to store the object in the internal in-memory database, not its full state.

You should only modify these parameters if you want to decrease the initial memory used by Connext DDS when a DomainParticipant is created or you want to increase the maximum number of local and remote Entities that can be stored in a DomainParticipant.

9.5.4.3 Example

For most applications, the default values for this QosPolicy may be sufficient. However, if an application uses the PARTITION, USER_DATA, TOPIC_DATA, or GROUP_DATA QosPolicies, the default maximum sizes of the data associated with those policies may need to be adjusted as required by the application. As noted previously, you must make sure that all DomainParticipants in the same DDS domain use the same sets of values or it is possible that Connext DDS will not successfully connect two Entities.

9.5.4.4 Properties

This QosPolicy cannot be modified after the DomainParticipant is created.

It can be set differently on the publishing and subscribing sides.

9.5.4.5 Related QosPolicies

- 9.5.1 DATABASE QosPolicy (DDS Extension) on page 637
- 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644
- 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432
- 7.5.30 USER_DATA QosPolicy on page 465
- 5.2.1 TOPIC_DATA QosPolicy on page 224
- 7.4.4 GROUP_DATA QosPolicy on page 353
- 7.4.5 PARTITION QosPolicy on page 356
- 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439

9.5.4.6 Applicable DDS Entities

- 9.3 DomainParticipants on page 609

9.5.4.7 System Resource Considerations

Memory and CPU usage are directly affected by the values set for parameters of this QosPolicy. See the detailed descriptions above for specifics.
9.5.5 EVENT QosPolicy (DDS Extension)

The EVENT QosPolicy configures the internal Connext DDS Event thread.

This QoS allows the you to configure thread properties such as priority level and stack size. You can also configure the maximum number of events that can be posted to the event thread. It contains the members in Table 9.14 DDS_EventQoSPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation.

### Table 9.14 DDS_EventQoSPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_ThreadSettings_t</td>
<td>thread.mask</td>
<td>Thread settings for the event thread used by Connext DDS to wake up for a timed event and possibly execute listener callbacks. The values used for these settings are OS-dependent; see the RTI Connext DDS Core Libraries Platform Notes for details. Note: thread.cpu_list and thread.cpu_rotation are not relevant in this QoS policy.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>initial_count</td>
<td>Initial number of events that can be stored simultaneously.</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>max_count</td>
<td>Maximum number of events that can be stored simultaneously.</td>
</tr>
</tbody>
</table>

The Event thread is used to wake up and execute timed events posted to the event queue. In a DomainParticipant, different Entities may have constraints that have to be checked at periodic intervals or at specific times. If the constraint is violated, a callback function may need to be executed. Timed events include checking for timeouts and deadlines, and executing internal and user timeout or exception handling routines/callbacks. A combination of a time, constraint, and callback can be considered to be an event. For more information, see 21.2 Event Thread on page 910.

For example, a DataReader may have a constraint that requires data to be received within a period of time specified by the 7.5.7 DEADLINE QosPolicy on page 405. For that DataReader, an event is stored by the Event thread so that it will wake up periodically to check to see if data has arrived in time. If not, the Event thread will execute the on_requested_deadline_missed() Listener callback of the DataReader (if it was installed and enabled).

A reliable connection between a DataWriter and DataReader will also post events for sending heartbeats used in the reliable protocol discussed in Reliable Communications (Chapter 11 on page 688).

This QoS configures the parameters associated with thread creation as well as the number of events that can be simultaneously stored by the Event thread.

#### 9.5.5.1 Example

In a real-time operating system, the priority of the Event thread should be set relative to the priority of the events that it must handle. For example, you may want the Event thread to have a high priority if the dead-
lines and callbacks that it handles are time or safety critical. It may be critical that the data of a particular DataReader arrives on time or if not, alternative action is taken with minimal latency.

If you create many Entities in a DomainParticipant with QosPolicies that will post events that check deadlines, liveliness or send heartbeats, then you may need to increase the maximum number of events that can be stored by the Event thread.

If your application is sending a lot of reliable data, you should increase the event thread priority to be higher than the sending thread priority.

9.5.5.2 Properties
This QosPolicy cannot be modified after the DomainParticipant is created.
It can be set differently on the publishing and subscribing sides.

9.5.5.3 Related QosPolicies
- 9.5.1 DATABASE QosPolicy (DDS Extension) on page 637
- 9.5.6 RECEIVER_POOL QosPolicy (DDS Extension) on the next page

9.5.5.4 Applicable DDS Entities
- 9.3 DomainParticipants on page 609

9.5.5.5 System Resource Considerations
Increasing initial_count and max_count will increase initial and maximum memory used for storing events.

Setting the thread parameters correctly on a real-time operating system is usually critical to the proper overall functionality of the applications on that system. Larger values for the thread.stack_size parameter will use up more memory.

By default, a DomainParticipant will dynamically allocate memory as needed for events posted to the event thread. However, by setting a maximum value or setting the initial and maximum value to be the same, you can either bound the amount of memory allocated for the event thread or prevent a DomainParticipant from dynamically allocating memory for the event thread after initialization.

9.5.6 RECEIVER_POOL QosPolicy (DDS Extension)
The RECEIVER_POOL QosPolicy configures the internal Connext DDS thread used to process the data received from a transport. The Receive thread is described in detail in 21.3 Receive Threads on page 911.

This QosPolicy contains the members in Table 9.15 DDS_ReceiverPoolQoSPolicy.
This QosPolicy sets the thread properties, like priority level and stack size, for the threads used to receive and process data from transports. *Connext DDS* uses a separate receive thread per port per transport plugin. To force *Connext DDS* to use a separate thread to process the data for a *DataReader*, you should set a unique port for the [TRANSPORT_UNICAST QosPolicy (DDS Extension)](page 460) or [TRANSPORT_MULTICAST QosPolicy (DDS Extension)](page 591) for the *DataReader*.

*Connext DDS* creates at least one thread for every transport that is installed and enabled for use by the *DomainParticipant* for receiving data. These threads are used to process data DDS samples received for the participant’s *DataReaders*, as well as messages used by *Connext DDS* itself in support of the application discovery process discussed in [Discovery (Chapter 15 on page 764)](index.html).

The user application may configure *Connext DDS* to create many more threads for receiving data sent via multicast or even to dedicate a thread to process the DDS data samples of a single *DataReader* received on a particular transport. This QosPolicy is used in the creation of all receive threads.

### 9.5.6.1 Example

When new data arrives on a transport, the receive thread may invoke the `on_data_available()` of the *Listener* callback of a *DataReader*. Thus, you may want to adjust the priority of the receive threads with
9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension)

respect to the other threads in the application as appropriate for the proper operation of the system.

9.5.6.2 Properties

This QosPolicy cannot be modified after the DomainParticipant is created.

It can be set differently on the publishing and subscribing sides.

9.5.6.3 Related QosPolicies

- 9.5.1 DATABASE QosPolicy (DDS Extension) on page 637
- 9.5.5 EVENT QosPolicy (DDS Extension) on page 662

9.5.6.4 Applicable DDS Entities

- 9.3 DomainParticipants on page 609

9.5.6.5 System Resource Considerations

Increasing the buffer_size will increase memory used by a receive thread.

Setting the thread parameters correctly on a real-time operating system is usually critical to the proper overall functionality of the applications on that system. Larger values for the thread.stack_size parameter will use up more memory.

9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension)

Connext DDS comes with three different transport plugins built into the core libraries (for most supported target platforms). These are plugins for UDPv4, shared memory (SHMEM), and UDPv6. (If you've installed RTI Real-Time WAN Transport, UDPv4_WAN is also available.)

This QosPolicy allows you to control which builtin transport plugins are used by a DomainParticipant. By default, only the UDPv4 and shared memory plugins are enabled (for most platforms; on some platforms, the shared memory plugin is not available). You can disable one or all of the builtin transports.

In some cases, users will disable the shared memory transport when they do not want applications to use shared memory to communicate when running on the same node.

This QoS Policy contains the member in Table 9.16 DDS_TransportBuiltinQosPolicy. For the default and valid values, please refer to the API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, QoS Policies, TRANSPORT_BUILTIN).
### 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension)

**Table 9.16 DDS_TransportBuiltinQosPolicy**

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TransportBuiltinKindMask</td>
<td>mask</td>
<td>A mask with bits that indicate which builtin transports will be installed. Three different transport plug-ins are built into the core Connext DDS libraries (for most supported target platforms): SHMEM, UDPv4, and UDPv6. (If you've installed Part 5: RTI Real-Time WAN Transport on page 1034, UDPv4_WAN is also available.) By default, the mask is set to SHMEM</td>
</tr>
</tbody>
</table>

You can set the mask programatically or via XML. For example, programatically:

```java
participant_qos.transport_builtin.mask = DDS_TRANSPORTBUILTIN_UDPv4 | DDS_TRANSPORTBUILTIN_SHMEM;
```

Via XML, you can use the `<transport_builtin>` tags. For example:

```xml
<domain_participant_qos>
  <transport_builtin>
    <mask>UDPv4</mask>
  </transport_builtin>
</domain_participant_qos>
```

In XML only, you can additionally configure the builtin transport properties, such as `<message_size_max>`. See 19.4.6 Transport Properties on page 880.

**Note:** Currently, Connext DDS will only listen for discovery traffic on the first multicast address (element 0) in `multicast_receive_addresses`.

### 9.5.7.1 Example

See 9.5.7.5 System Resource Considerations on the next page for an example of why you may want to use this QosPolicy.

In addition, customers may wish to install and use their own custom transport plugins instead of any of the builtin transports. In that case, this QosPolicy may be used to disable all builtin transports.

### 9.5.7.2 Properties

This QosPolicy cannot be modified after the `DomainParticipant` is created.

It can be set differently on the publishing and subscribing sides.

### 9.5.7.3 Related QosPolicies

- 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459
- 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460
9.5.8 TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension)

- 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591

9.5.7.4 Applicable DDS Entities

- 9.3 DomainParticipants on page 609

9.5.7.5 System Resource Considerations

You can save memory and other system resources if you disable the built-in transports that your application will not use. For example, if you only run a single application with a single DomainParticipant on each machine in your network, then you can disable the shared memory transport since your applications will never use it to send or receive messages.

9.5.8 TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension)

The multicast address on which a DataReader wants to receive its data can be explicitly configured using the 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591. However in systems with many multicast addresses, managing the multicast configuration can become cumbersome. The TransportMulticastMapping QosPolicy is designed to make configuration and assignment of the DataReader's multicast addresses more manageable. When using this QosPolicy, the middleware will automatically assign a multicast receive address for a DataReader from a range by using configurable mapping rules.

DataReaders can be assigned a single multicast receive address using the rules defined in this QosPolicy on the DomainParticipant. This multicast receive address is exchanged during simple discovery in the same manner used when the multicast receive address is defined explicitly. No additional configuration on the writer side is needed.

Mapping within a range is done through a mapping function. The middleware provides a default hash (md5) mapping function. This interface is also pluggable, so you can specify a custom mapping function to minimize collisions.

To use this QosPolicy, you must set the kind in the 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591 to AUTOMATIC.

This QosPolicy contains the member in Table 9.17 DDS_TransportMulticastMappingQosPolicy.

### Table 9.17 DDS_TransportMulticastMappingQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TransportMapping</td>
<td>SettingsSeq</td>
<td>A sequence of multicast communication settings, each of which has the format shown in Table 9.18 DDS_TransportMulticastSettings_t.</td>
</tr>
</tbody>
</table>
### 9.5.8 TRANSPORT_MULTICAST_MAPPING QosPolicy (DDS Extension)

#### Table 9.18 DDS_TransportMulticastSettings_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char*</td>
<td>addresses</td>
<td>A string containing a comma-separated list of IP addresses or IP address ranges to be used to receive multicast traffic for the entity with a topic that matches the <code>topic_expression</code>. See 9.5.8.1 Formatting Rules for Addresses on the next page.</td>
</tr>
<tr>
<td>char*</td>
<td>topic_expression</td>
<td>A regular expression used to map topic names to corresponding addresses. See 5.4.6.5 SQL Extension: Regular Expression Matching on page 241.</td>
</tr>
<tr>
<td>DDS_TransportMulticastMappingFunction_t</td>
<td>mapping_function</td>
<td>Optional. Defines a user-provided pluggable mapping function. See Table 9.19 DDS_TransportMulticastMappingFunction_t.</td>
</tr>
</tbody>
</table>

#### Table 9.19 DDS_TransportMulticastMappingFunction_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char*</td>
<td>dll</td>
<td>Specifies a dynamic library that contains a mapping function. You may specify a relative or absolute path. If the name is specified as &quot;foo&quot;, the library name on Linux systems will be <code>libfoo.so</code>; on Windows systems it will be <code>foo.dll</code>.</td>
</tr>
<tr>
<td>char*</td>
<td>function_name</td>
<td>Specifies the name of a mapping function in the library specified in the above dll. The function must implement the following interface: [\text{int function(const char* topic_name, int numberOfAddresses);}] The function must return an integer that indicates the index of the address to use for the given <code>topic_name</code>. For example, if the first address in the list should be used, it must return 0; if the second address in the list should be used, it must return 1, etc.</td>
</tr>
</tbody>
</table>

#### 9.5.8.1 Formatting Rules for Addresses

- The string must contain IPv4 or IPv6 addresses separated by commas. For example: "239.255.100.1,239.255.100.2,239.255.100.3"
- You may specify ranges of addresses by enclosing the start and end addresses in square brackets. For example: "[239.255.100.1,239.255.100.3]".  
- You may combine the two approaches. For example: "239.255.200.1,[239.255.100.1,239.255.100.3], 239.255.200.3"  
- IPv4 addresses must be specified in Dot-decimal notation.  
- IPv6 addresses must be specified using 8 groups of 16-bit hexadecimal values separated by colons. For example: FF00:0000:0000:0000:0202:B3FF:FE1E:8329.
Leading zeroes can be skipped. For example: FF00:0:0:202:B3FF:FE1E:8329.

You may replace a consecutive number of zeroes with a double colon, but only once within an address. For example: FF00::202:B3FF:FE1E:8329.

### 9.5.8.2 Example

This QoS policy configures the multicast ranges and mapping rules at the DomainParticipant level. You can configure a large set of multicast addresses on the DomainParticipant.

In addition, you can configure a mapping between topic names and multicast addresses. For example, topic "A" can be assigned to address 239.255.1.1 and topic "B" can be assigned to address 239.255.1.2.

This configuration is quite flexible. For example, you can specify mappings between a subset of topics to a range of multicast addresses. For example, topics "X", "Y" and "Z" can be mapped to [239.255.1.1, 239.255.1.255], or using regular expressions, "X*" and "B-Z" can be mapped to a sub-range of addresses. See 5.4.6.5 SQL Extension: Regular Expression Matching on page 241.

### 9.5.8.3 Properties

This QosPolicy cannot be modified after the DomainParticipant is created.

### 9.5.8.4 Related QosPolicies

- 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591

### 9.5.8.5 Applicable DDS Entities

- 9.3 DomainParticipants on page 609

### 9.5.8.6 System Resource Considerations

See 8.6.5.5 System Resource Considerations on page 594.

### 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension)

The WIRE_PROTOCOL QosPolicy configures some global Real-Time Publish Subscribe (RTPS) protocol-related properties for the DomainParticipant. The RTPS OMG-standard, interoperability protocol is used by Connext DDS to format and interpret messages between DomainParticipants.

It includes the members in Table 9.20 DDS_WireProtocolQosPolicy. For defaults and valid ranges, please refer to the API Reference HTML documentation. (The default values contain the correctly initialized wire protocol attributes. They should not be modified without an understanding of the underlying Real-Time Publish Subscribe (RTPS) wire protocol.)
### Table 9.20 DDS_WireProtocolQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>participant_id</td>
<td>Unique identifier for participants that belong to the same DDS domain on the same host. See 9.5.9.1 Choosing Participant IDs on the next page.</td>
</tr>
<tr>
<td>DDS_UnsignedLong</td>
<td>rtps_host_id</td>
<td>A machine/OS-specific host ID, unique in the DDS domain. See 9.5.9.3 Controlling How the GUID is Set (rtps_auto_id_kind) on page 673.</td>
</tr>
<tr>
<td></td>
<td>rtps_app_id</td>
<td>A participant-specific ID, unique within the scope of the rtps_host_id. See 9.5.9.3 Controlling How the GUID is Set (rtps_auto_id_kind) on page 673.</td>
</tr>
<tr>
<td></td>
<td>rtps_instance_id</td>
<td>An instance-specific ID of the DomainParticipant that, together with the rtps_app_id, is unique within the scope of the rtps_host_id. See 9.5.9.3 Controlling How the GUID is Set (rtps_auto_id_kind) on page 673.</td>
</tr>
<tr>
<td>DDS_RtpsWellKnownPorts_t</td>
<td>rtps_well_known_ports</td>
<td>Determines the well-known multicast and unicast ports for discovery and user traffic. See 9.5.9.2 Ports Used for Discovery on page 672.</td>
</tr>
<tr>
<td>DDS_RtpsReservedPortKindMask</td>
<td>rtps_reserved_ports_mask</td>
<td>Specifies which well-known multicast and unicast ports to reserve when enabling the DomainParticipant.</td>
</tr>
<tr>
<td>DDS_WireProtocolQosPolicyAutoKind</td>
<td>rtps_auto_id_kind</td>
<td>Kind of auto mechanism used to calculate the GUID prefix.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>compute_crc</td>
<td>Adds an RTPS CRC submessage to every message.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>check_crc</td>
<td>Checks if the received RTPS message is valid by comparing the computed CRC with the received RTPS CRC submessage.</td>
</tr>
</tbody>
</table>

Note that 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 and 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570 configure RTPS and reliability properties on a per DataWriter and DataReader basis.

### 9.5.9.1 Choosing Participant IDs

When you create a DomainParticipant, you must specify a domain ID, which identifies the communication channel across the whole system. Each DomainParticipant in the same DDS domain on the same host also needs a unique integer, known as the participant_id.

The participant_id uniquely identifies a DomainParticipant from other DomainParticipants in the same DDS domain on the same host. You can use the same participant_id value for DomainParticipants in the same DDS domain but running on different hosts.

The participant_id is also used to calculate the default unicast user-traffic and the unicast meta-traffic port numbers, as described in 15.5 Ports Used for Discovery on page 791. If you only have one DomainParticipant in the same DDS domain on the same host, you will not need to modify this value.
You can either allow Connext DDS to select a participant ID automatically (by setting `participant_id` to -1), or choose a specific participant ID (by setting `participant_id` to the desired value).

- **Automatic Participant ID Selection**

  The default value of `participant_id` is -1, which means Connext DDS will select a participant ID for you.

  Connext DDS will pick the smallest participant ID, based on the unicast ports available on the transports enabled for discovery, based on the unicast and/or multicast ports available on the transports enabled for discovery and/or user traffic.

  The `rtps_reserved_ports_mask` field determines which ports to check when picking the next available participant ID. The reserved ports are calculated based on the formula specified in 15.5.1 Inbound Ports for Meta-Traffic on page 793 and 15.5.2 Inbound Ports for User Traffic on page 793. By default, Connext DDS will reserve the meta-traffic unicast port, the meta-traffic multicast port, and the user traffic unicast port.

  Connext DDS will attempt to resolve an automatic port ID either when a `DomainParticipant` is enabled, or when a `DataReader` or a `DataWriter` is created. Therefore, all the transports enabled for discovery must have been registered by this time. Otherwise, the discovery transports registered after resolving the automatic port index may produce port conflicts when the `DomainParticipant` is enabled.

  To see what value Connext DDS has selected, either:
  - Change the verbosity level of the `NDDS_CONFIG_LOG_CATEGORY_API` category to `NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL` (see 25.2 Controlling Messages from Connext DDS on page 984).
  - Call `get_qos()` and look at the `participant_id` value in the 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on the previous page after the `DomainParticipant` is enabled.

- **Manual Participant ID Selection**

  If you do have multiple `DomainParticipants` on the same host, you should use consecutively numbered participant indices start from 0. This will make it easier to specify the discovery peers using the `initial_peers` parameter of this QosPolicy or the `NDDS_DISCOVERY_PEERS` environment variable. See 15.2 Configuring the Peers List Used in Discovery on page 767 for more information.

  Do not use random participant indices since this would make DISCOVERY incredibly difficult to configure. In addition, the `participant_id` has a maximum value of 120 (and will be less for domain IDs other than 0) when using an IP-based transport since the `participant_id` is used to create the port number (see 15.5 Ports Used for Discovery on page 791), and for IP, a port number cannot be larger than 65536.
For details, see 15.5 Ports Used for Discovery on page 791.

### 9.5.9.2 Ports Used for Discovery

The `rtps_well_known_ports` structure allows you to configure the ports that are used for discovery of inbound meta-traffic (discovery data internal to `Connext DDS`) and user traffic (from your application).

It includes the members in Table 9.21 DDS_RtpsWellKnownPorts_t. For defaults and valid ranges, please refer to the API Reference HTML documentation.

#### Table 9.21 DDS_RtpsWellKnownPorts_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long</td>
<td>port_base</td>
<td>The base port offset. All mapped well-known ports are offset by this value. Resulting ports must be within the range imposed by the underlying transport.</td>
</tr>
<tr>
<td></td>
<td>domain_id_gain</td>
<td>Tunable gain parameters. See 15.5 Ports Used for Discovery on page 791.</td>
</tr>
<tr>
<td></td>
<td>participant_id_gain</td>
<td>Tunable gain parameters. See 15.5 Ports Used for Discovery on page 791.</td>
</tr>
<tr>
<td></td>
<td>builtin_multicast_port_offset</td>
<td>Additional offset for meta-traffic port. See 15.5.1 Inbound Ports for Meta-Traffic on page 793.</td>
</tr>
<tr>
<td></td>
<td>builtin_unicast_port_offset</td>
<td>Additional offset for meta-traffic port. See 15.5.1 Inbound Ports for Meta-Traffic on page 793.</td>
</tr>
<tr>
<td></td>
<td>user_multicast_port_offset</td>
<td>Additional offset for user traffic port. See 15.5.2 Inbound Ports for User Traffic on page 793.</td>
</tr>
<tr>
<td></td>
<td>user_unicast_port_offset</td>
<td>Additional offset for user traffic port. See 15.5.2 Inbound Ports for User Traffic on page 793.</td>
</tr>
</tbody>
</table>

### 9.5.9.3 Controlling How the GUID is Set (`rtps_auto_id_kind`)

In order for the discovery process to work correctly, each `DomainParticipant` must have a unique identifier. This QoS policy specifies how that identifier should be generated.

RTPS defines a 96-bit prefix to this identifier; each `DomainParticipant` must have a unique value of this prefix relative to all other participants in its DDS domain.

If an application dies unexpectedly and is restarted, the IDs used by the new instance of `DomainParticipants` should be different than the ones used by the previous instances. A change in these values allows other `DomainParticipants` to know that they are communicating with a new instance of an application, and not the previous instance.

For legacy reasons, `Connext DDS` divides the 96-bit prefix into three integers:

- The first integer is called **host ID**. The original purpose of this integer was to contain the identity of the machine on which the `DomainParticipant` is executing.
• The second integer is called an **application ID**. The original purpose of this integer was to contain a value that identifies the process or task in which the *DomainParticipant* is contained.  

• The third integer is called **instance ID**. The original purpose was to contain a value that uniquely identifies a *DomainParticipant* within a task or process.

The `rtps_auto_id_kind` field can be used to configure the algorithm that *Connext DDS* uses to populate the 96-bit prefix. Then you can optionally overwrite specific parts of the 96-bit prefix by explicitly configuring the `rtps_host_id` (first integer), `rtps_app_id` (second integer), and `rtps_instance_id` (third integer).

The `rtps_auto_id_kind` field supports three different prefix generation algorithms:

1. In the default and most common scenario, `rtps_auto_id_kind` is set to RTPS/AUTO_ID_FROM_UUID. As the name suggests, this mechanism uses a unique, randomly generated UUID to fill the `rtps_host_id`, `rtps_app_id`, or `rtps_instance_id` fields. The first two bytes of the `rtps_host_id` are replaced with the RTI vendor ID (0x0101).

2. (Legacy) When `rtps_auto_id_kind` is set to DDS/RTPS/AUTO_ID_FROM_IP, the 96-bit prefix is generated as follows:
   - `rtps_host_id`: the 32 bit value of the IPv4 of the first up and running interface of the host machine is assigned. If the host does not have an IPv4 address, the host-id will be automatically set to 0x7F000001.
   - `rtps_app_id`: the process (or task) ID is assigned.
   - `rtps_instance_id`: A counter is assigned that is incremented per new participant within a process.

   DDS/RTPS/AUTO_ID_FROM_IP is not a good algorithm to guarantee prefix uniqueness, because the process ID can be recycled by the OSs. See 9.5.9.3.2 Uniqueness Problem with DDS/RTPS/AUTO_ID_FROM_IP and DDS/RTPS/AUTO_ID_FROM_MAC on page 676 for additional information.

3. (Legacy) When `rtps_auto_id_kind` is set to DDS/RTPS/AUTO_ID_FROM_MAC, the 96-bit prefix is generated as follows:
   - `rtps_host_id`: the first 32 bits of the MAC address of the first up and running interface of the host machine are assigned.
   - `rtps_app_id`: the last 32 bits of the MAC address of the first up and running interface of the host machine are assigned.
   - `rtps_instance_id`: this field is split into two different parts. The process (or task) ID is assigned to the first 24 bits. A counter is assigned to the last 8 bits. This counter is incremented per new participant. In both scenarios, you can change the value of each field independently.
DDS_RTPS_AUTO_ID_FROM_IP is not a good algorithm to guarantee prefix uniqueness because the process ID can be recycled by the OSs. See 9.5.9.2 Uniqueness Problem with DDS_RTPS_AUTO_ID_FROM_IP and DDS_RTPS_AUTO_ID_FROM_MAC on page 676 for additional information.

### 9.5.9.3.1 Overwriting the Default RTPS 96-bit Prefix

Some examples are provided to better explain the behavior of this QoSPolicy in case you want to change the default behavior with DDS_RTPS_AUTO_ID_FROM_MAC.

1. Get the *DomainParticipant* QoS from the DomainParticipantFactory:

   ```c
   DDS_DomainParticipantFactory_get_default_participant_qos(
       DDS_DomainParticipantFactory_get_instance(),
       &participant_qos);
   ```

2. Change the WireProtocolQosPolicy using one of the following options.

   - Use DDS_RTPS_AUTO_ID_FROM_MAC to explicitly set just the application/task identifier portion of the `rtps_instance_id` field:

     ```c
     participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
     participant_qos.wire_protocol.rtps_host_id = DDS_RTPS_AUTO_ID;
     participant_qos.wire_protocol.rtps_app_id = DDS_RTPS_AUTO_ID;
     participant_qos.wire_protocol.rtps_instance_id =
          /* App ID */ (12 << 8) |
          /* Instance ID*/ (DDS_RTPS_AUTO_ID);
     ```

   - Only set the per participant counter and let Connext DDS handle the application/task identifier:

     ```c
     participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
     participant_qos.wire_protocol.rtps_host_id = DDS_RTPS_AUTO_ID;
     participant_qos.wire_protocol.rtps_app_id = DDS_RTPS_AUTO_ID;
     participant_qos.wire_protocol.rtps_instance_id =
          /* App ID */ (DDS_RTPS_AUTO_ID) |
          /* Instance ID*/ (12));
     ```

   - Set the entire `rtps_instance_id` field yourself:

     ```c
     participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
     participant_qos.wire_protocol.rtps_host_id = DDS_RTPS_AUTO_ID;
     participant_qos.wire_protocol.rtps_app_id = DDS_RTPS_AUTO_ID;
     participant_qos.wire_protocol.rtps_instance_id =
          /* App ID */ (DDS_RTPS_AUTO_ID) |
          /* Instance ID*/ (id));
     ```
### 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension)

```c
participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
participant_qos.wire_protocol.rtps_host_id = DDS_RTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_app_id = DDS_RTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_instance_id =
  ( /* App ID */ (12 << 8)) |
  /* Instance ID */ (9 )
```

**Note:** If you are using DDS_RTPS_AUTO_ID_FROM_MAC as *rtps_auto_id_kind* and you decide to manually handle the *rtps_instance_id* field, you must ensure that both parts are non-zero (otherwise *Connext DDS* will take responsibility for them).

*RTI* recommends that you always specify the two parts separately in order to avoid errors.

- Let *Connext DDS* handle the entire *rtps_instance_id* field:

```c
participant_qos.wire_protocol.rtps_auto_id_kind = DDS_RTPS_AUTO_ID_FROM_MAC;
participant_qos.wire_protocol.rtps_host_id = DDS_RTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_app_id = DDS_RTPS_AUTO_ID;
participant_qos.wire_protocol.rtps_instance_id = DDS_RTPS_AUTO_ID;
```

**Note:** If you are using DDS_RTPS_AUTO_ID_FROM_MAC as *rtps_auto_id_kind* and you decide to manually set the *rtps_instance_id* field, you must ensure that both parts are non-zero (otherwise *Connext DDS* will take responsibility for them). *RTI* recommends that you always specify the two parts separately in order to clearly show the difference.

3. Create the *DomainParticipant* as usual using the modified QoS structure instead of the default one.

#### 9.5.9.3.2 Uniqueness Problem with DDS_RTPS_AUTO_ID_FROM_IP and DDS_RTPS_AUTO_ID_FROM_MAC

This section applies only when *rtps_auto_id_kind* is set to DDS_RTPS_AUTO_ID_FROM_IP or DDS_RTPS_AUTO_ID_FROM_MAC.

On many real-time operating systems, and even on some non-real-time operating systems, when a node is rebooted, and applications are automatically started, process IDs are deterministically assigned. That is, when the system restarts or if an application dies and is restarted, the application will be reassigned the same process or task ID.

This means that *Connext DDS*’s automatic algorithm for creating unique *rtps_app_id*’s will produce the same value between sequential instances of the same application. This will confuse the other *DomainParticipants* on the network into thinking that they are communicating with the previous instance of the application instead of a new instance. Errors usually resulting in a failure to communicate will ensue.
Thus, for applications running on nodes that may be rebooted without letting the application shutdown appropriately (destroying the DomainParticipant), especially on nodes running real-time operating systems like VxWorks, you will want to set the rtps_app_id manually. We suggest that a strictly incrementing counter is stored either on a file system or in non-volatile RAM for the rtps_app_id.

Whatever method you use, you should make sure that the rtps_app_id is unique across all DomainParticipants running on a host as well as DomainParticipants that were recently running on the host. After a period configured through the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644, existing applications will eventually flush old DomainParticipants that did not properly shutdown from their databases. When that is done, then rtps_app_id may be reused.

### 9.5.9.4 Properties

This QosPolicy cannot be modified after the DomainParticipant is created.

If manually set, it must be set differently for every DomainParticipant in the same DDS domain across all applications. The value of rtps_app_id should also change between different invocations of the same application (for example, when an application is restarted).

### 9.5.9.5 Related QosPolicies

- 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644

### 9.5.9.6 Applicable DDS Entities

- 9.3 DomainParticipants on page 609

### 9.5.9.7 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

### 9.6 Clock Selection

Connext DDS uses clocks to measure time and generate timestamps.

The middleware uses two clocks: an internal clock and an external clock.

- The internal clock measures time and handles all timing in the middleware.
- The external clock is used solely to generate timestamps (such as the source timestamp and the reception timestamp), in addition to providing the time given by the DomainParticipant’s get_current_time() operation (see 9.3.15.2 Getting the Current Time on page 631).

### 9.6.1 Available Clocks

Two clock implementations are generally available: the real-time clock and the monotonic clock.
The real-time clock provides the real time of the system. This clock may generally be monotonic, but may not be guaranteed to be so. It is adjustable and may be subject to small and large changes in time. The time obtained from this clock is generally a meaningful time, in that it is the amount of time from a known epoch. For the purposes of clock selection, this clock can be referenced by the names "realtime" or "system"—both names map to the same real-time clock.

The monotonic clock provides times that are monotonic from a clock that is not adjustable. This clock is not subject to changes in the system or realtime clock, which may be adjusted by the user or via time synchronization protocols. However, this clock’s time generally starts from an arbitrary point in time, such as system start-up. Note that the monotonic clock is not available for all architectures. Please see the RTI Connext DDS Core Libraries Platform Notes for the architectures on which it is supported. For the purposes of clock selection, this clock can be referenced by the name "monotonic".

### 9.6.2 Clock Selection Strategy

To configure the clock selection, use the DomainParticipant’s 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439. Table 9.22 Clock Selection Properties lists the supported properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.clock.external_clock</td>
<td>Comma-delimited list of clocks to use for the external clock, in the order of preference. Valid clock names are “realtime”, “system”, or “monotonic”.</td>
</tr>
<tr>
<td>dds.clock.internal_clock</td>
<td>Comma-delimited list of clocks to use for the internal clock, in the order of preference. Valid clock names are “realtime”, “system”, or “monotonic”.</td>
</tr>
</tbody>
</table>

By default, both the internal and external clocks use the realtime clock.

If you want your application to be robust to changes in the system time, you may use the monotonic clock as the internal clock, and leave the system clock as the external clock. However, note that this may slightly diminish performance, in that both the send and receive paths may need to get times from both clocks.

Since the monotonic clock is not available on all architectures, you may want to specify "monotonic, realtime" for the internal_clock property (see Table 9.22 Clock Selection Properties). By doing so, the middleware will attempt to use the monotonic clock if it is available, and will fall back to the realtime clock if the monotonic clock is not available.

If you want the application to be robust to changes in the system time, you are not relying on source timestamps, and you want to avoid obtaining times from both clocks, you may use the monotonic clock for both the internal and external clocks.
9.7 System Properties

Connext DDS uses the DomainParticipant’s PropertyQosPolicy to maintain a set of properties that provide system information, such as the hostname.

Unless the default the DDS_DomainParticipantQos structure (see 9.3.7 Setting DomainParticipant QosPolicies on page 622) is overwritten, the system properties are automatically set in the DDS_DomainParticipantQos structure that is obtained by calling the DomainParticipantFactory’s get_default_participant_qos() operation or by using the constant DDS_PARTICIPANT_QOS_DEFAULT.

System properties are also automatically set in the DDS_DomainParticipantQos structure loaded from an XML QoS profile unless you disable property inheritance using the attribute inherit in the XML tag <property>.

By default, the system properties are propagated to other DomainParticipants in the system and can be accessed through the property field in the Table 18.1 Participant Built-in Topic’s Data Type (DDS_ParticipantBuiltinTopicData).

You can disable propagation of individual properties by setting the property’s propagate flag to FALSE or by removing the property using the PropertyQosPolicyHelper operation, remove_property() (see Table 7.65 PropertyQoSPolicyHelper Operations).

The number of system properties that are initialized for a DomainParticipant is platform specific: only process_id and os_arch are supported on all platforms.

These properties will only be created if Connext DDS can obtain the information for them; see Table 9.23 System Properties.

System properties are affected by the DomainParticipantResourceLimitsQosPolicy’s participant_property_list_max_length and participant_property_string_max_length.

Table 9.23 System Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.sys_info.creation_timestamp</td>
<td>Time when the executable was created.¹</td>
</tr>
<tr>
<td>dds.sys_info.executable_filepath</td>
<td>Name and full path of the executable.²</td>
</tr>
<tr>
<td>dds.sys_info.execution_timestamp</td>
<td>Time when the execution started.³</td>
</tr>
</tbody>
</table>

¹Only supported on Windows, INtime®, and Linux architectures.

²Only supported on Windows and Linux architectures.

³Only supported on Windows, INtime, and Linux architectures.
### Table 9.23 System Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.sys_info.hostname</td>
<td>Hostname(^1)</td>
</tr>
<tr>
<td>dds.sys_info.target</td>
<td>Architecture for which the library was compiled (for example, x64Darwin10gcc4.2.1).</td>
</tr>
<tr>
<td>dds.sys_info.process_id</td>
<td>Process ID</td>
</tr>
<tr>
<td>dds.sys_info.username</td>
<td>Username that is running the process.(^2)</td>
</tr>
</tbody>
</table>

\(^1\)Supported on Windows, Linux, macOS, INtime, and QNX architectures.

\(^2\)Only supported on Windows, INtime, and Linux architectures.
Chapter 10 Building Applications

This chapter provides instructions on how to build Connext DDS applications for the following platforms:

- 10.4 Linux Platforms on page 683
- 10.5 Windows Platforms on page 684
- 10.6 Java Platforms on page 685

While you can create applications for other operating systems, the platforms presented in this chapter are a good starting point. We recommend that you first build and test your application on one of these systems.

Instructions for other supported target platforms are provided in the RTI Connext DDS Core Libraries Platform Notes.

To build a non-Java application using Connext DDS, you must specify the following items:

- NDDSHOME environment variable
- Connext DDS header files
- Connext DDS libraries to link
- Compatible system libraries
- Compiler options

To build Java applications using Connext DDS, you must specify the following items:

- NDDSHOME environment variable
- Connext DDS JAR file
10.1 Running on a Computer Not Connected to a Network

- Compatible Java virtual machine (JVM)
- Compiler options

This chapter describes the basic steps you will take to build an application on the above-mentioned platforms. Specific details, such as exactly which libraries to link, compiler flags, etc. are in the RTI Connext DDS Core Libraries Platform Notes.

10.1 Running on a Computer Not Connected to a Network

If you want to run Connext DDS applications on the same computer, and that computer is not connected to a network, you must set NDDS_DISCOVERY_PEERS so that it will only use shared memory. For example:

```
set NDDS_DISCOVERY_PEERS=4@shm:////
```

(The number 4 is only an example. This is the maximum participant ID.)

10.2 Connext DDS Header Files — All Architectures

You must include the appropriate Connext DDS header files, which are listed in Table 10.1 Header Files to Include for Connext DDS (All Architectures). The header files that need to be included depend on the API being used.

**Table 10.1 Header Files to Include for Connext DDS (All Architectures)**

<table>
<thead>
<tr>
<th>Connext DDS API</th>
<th>Header Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>#include &quot;ndds/ndds_c.h&quot;</td>
</tr>
<tr>
<td>C++</td>
<td>#include &quot;ndds/ndds_cpp.h&quot;</td>
</tr>
<tr>
<td>C++/CLI, C#, Java</td>
<td>none</td>
</tr>
</tbody>
</table>

For the compiler to find the included files, the path to the appropriate include directories must be provided. Table 10.2 Include Paths for Compilation (All Architectures) lists the appropriate include path for use with the compiler. The exact path depends on where you installed Connext DDS. See Paths Mentioned in Documentation on page 1.

**Table 10.2 Include Paths for Compilation (All Architectures)**

<table>
<thead>
<tr>
<th>Connext DDS API</th>
<th>Include Path Directories</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>&lt;$NDDSHOME&gt;/include&lt;br/&gt; &lt;$NDDSHOME&gt;/include/ndds</td>
</tr>
<tr>
<td>C++/CLI, C#, Java</td>
<td>none</td>
</tr>
</tbody>
</table>
10.3 Choosing the Right Libraries

The header files that define the data types you want to use within the application also need to be included. For example, Table 10.3 Header Files to Include for Data Types (All Architectures) lists the files to be include for type “Foo” (these are the filenames generated by RTI Code Generator, described in Data Types and DDS Data Samples (Chapter 3 on page 27)).

<table>
<thead>
<tr>
<th>Connext DDS API</th>
<th>User Data Type Header Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>C and C++</td>
<td>#include &quot;Foo.h&quot;</td>
</tr>
<tr>
<td></td>
<td>#include &quot;FooSupport.h&quot;</td>
</tr>
<tr>
<td>C++/CLI, C#, Java</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 10.3 Header Files to Include for Data Types (All Architectures)

10.3 Choosing the Right Libraries

10.3.1 Required Libraries

All required system and Connext DDS libraries are listed in the RTI Connext DDS Core Libraries Platform Notes.

You must choose between dynamic (shared) and static libraries. Do not mix the different types of libraries during linking. The benefit of linking against the dynamic libraries is that your final executables’ sizes will be significantly smaller. You will also use less memory when you are running several Connext DDS applications on the same node. However, shared libraries require more setup and maintenance during upgrades and installations.

To see if dynamic libraries are supported for your target architecture, see the RTI Connext DDS Core Libraries Platform Notes.1

10.3.2 Mixing Static and Dynamic Libraries not Supported

You must choose either static or dynamic linking. Mixing static and dynamic RTI libraries—for example, using RTI static core libraries and dynamic TCP Transport—is not supported.

The examples in this section are for Linux systems, but except for small differences in names, the same concepts apply to Windows and macOS systems.

Suppose you have a Connext DDS-based application myApp, and you want to use the TCP Transport plugin. The library dependency looks something like that in Figure 10.1: Library Dependency on the next page.

---

1In the Platform Notes, see the “Building Instructions...” table for your target architecture.
10.4 Linux Platforms

Before building a Connext DDS application for a Linux platform (including QNX® systems), make sure that:

- A supported version of your architecture is installed. See the RTI Connext DDS Core Libraries Platform Notes for supported architectures.
- Connext DDS 6.x.y is installed (where 6.x.y stands for the version number of the current release). For installation instructions, refer to the RTI Connext DDS Installation Guide.
- A “make” tool is installed. RTI recommends GNU Make. If you do not have it, you may be able to download it from your operating system vendor. Learn more at www.gnu.org/software/make/ or download from ftpmirror.gnu.org/make as source code.
- The NDDSHOME environment variable is set to the root directory of the Connext DDS installation (such as /home/user/rti_connexx_dds-6.x.y).
  - To confirm, type this at a command prompt:

    ```bash
    echo $NDDSHOME
    env | grep NDDSHOME
    ```
  - If it is not set or is set incorrectly, type:

    ```bash
    setenv NDDSHOME <correct directory>
    ```
To compile a *Connext DDS* application of any complexity, either modify the auto-generated makefile created by running *RTI Code Generator* or write your own makefile. See the *RTI Connext DDS Core Libraries Platform Notes* for information on compiler flags.

**10.5 Windows Platforms**

Before building an application for a Microsoft Windows® platform, make sure that:

- Supported versions of Windows and Visual Studio are installed. See the Windows section of the *RTI Connext DDS Core Libraries Platform Notes*.
- *Connext DDS* 6.x.y is installed (where 6.x.y stands for the version numbers of the current release). For installation instructions, refer to the *RTI Connext DDS Installation Guide*.
- The NDDSHOME environment variable is set to the root directory of the *Connext DDS* installation (such as C:\Program Files\rti_connext_dds-6.x.y). To confirm, type this at a command prompt:

  ```
echo %NDDSHOME%
  ```

- You use the *dynamic* MFC Library (not static).

  To avoid communication problems in your *Connext DDS* application, use the dynamic MFC library, not the static version. (If you use the static version, your *Connext DDS* application may stop receiving DDS samples once the Windows sockets are initialized.)

To compile a *Connext DDS* application of any complexity, use a project file in Microsoft Visual Studio. The project settings are described below. The Windows section of the *RTI Connext DDS Core Libraries Platform Notes* contains more information.

**10.5.1 Using Visual Studio**

1. Select the multi-threaded project setting:
   a. From the Project menu, select Properties.
   b. Select the C/C++ folder.
   c. Select Code Generation.
   d. Set the Runtime Library field to one of the options from Table 10.4 Runtime Library Settings for Visual Studio.

2. Link against the *Connext DDS* libraries:
   a. Select the Linker folder on the Project, Properties dialog box.
   b. Select the Input properties.
   c. See the Windows section of the *RTI Connext DDS Core Libraries Platform Notes* for a list of required libraries. You have a choice of whether to link with *Connext DDS*’s static or
dynamic libraries. Decide whether or not you want debugging symbols on. In either case, be sure to use a space as a delimiter between libraries, not a comma. Add the libraries to the beginning of the Additional Dependencies field.

d. Select the General properties.
e. Add the following to the Additional library path field (replace <architecture> to match your installed system):

\$(NDDS_HOME)/lib/<architecture>

3. Specify the path to Connext DDS’s header file:
   a. Select the C/C++ folder.
   b. Select the General properties.
   c. In the Additional include directories: field, add paths to the “include” and “include\ndds” directories. For example: (your paths may differ, depending on where you installed Connext DDS).

   c:\Program Files\rti_connext_dds-6.x.y\include\nc:\Program Files\rti_connext_dds-6.x.y\include\ndds

<table>
<thead>
<tr>
<th>If you are using this Library Format...</th>
<th>Set the Runtime Library field to...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Release version of static libraries</td>
<td>Multi-threaded DLL (/MD)</td>
</tr>
<tr>
<td>Debug version of static libraries</td>
<td>Multi-threaded Debug DLL (/MDd)</td>
</tr>
<tr>
<td>Release version of dynamic libraries</td>
<td>Multi-threaded DLL (/MD)</td>
</tr>
<tr>
<td>Debug version of dynamic libraries</td>
<td>Multi-threaded Debug DLL (/MDd)</td>
</tr>
</tbody>
</table>

**Table 10.4 Runtime Library Settings for Visual Studio**

### 10.6 Java Platforms

Before building an application for a Windows or Linux Java platform, make sure that:

- *Connext DDS 6.x.y* is installed (where 6.x.y stands for the version numbers of the current release).
- A supported version of a JDK is installed. See the appropriate section of the **RTI Connext DDS Core Libraries Platform Notes**.

#### 10.6.1 Java Libraries

*Connext DDS* requires that certain Java archive (JAR) files be on your classpath when running *Connext DDS* applications. See the **RTI Connext DDS Core Libraries Platform Notes** for more details.
10.6.2 Native Libraries

Connext DDS for Java is implemented using Java Native Interface (JNI), so it is necessary to provide your Connext DDS distributed applications access to certain native shared libraries. See the RTI Connext DDS Core Libraries Platform Notes for more details.

10.7 Building Applications Using CMake

Connext DDS allows you to integrate the RTI Connext DDS libraries with build systems implemented using CMake®. A “Find Package” CMake script is provided as part of the Connext DDS installation. That script helps the build system find all the RTI Connext DDS libraries and include directories needed by your application. So, instead of setting the variables manually in your CMake scripts, you can call the Connext DDS “Find Package CMake” script to set all the variables needed by your application.

You can find the script (FindRTIConnextDDS.cmake) in <NDDSHOME>/resource/cmake. To know more about the input and output variables, see the documentation included in the script.

See the RTI Connext DDS Core Libraries Platform Notes for the platforms that support this script.
Part 3: Advanced Concepts

This part of the manual will guide you through some of the more advanced concepts:

- Reliable Communications (Chapter 11 on page 688)
- Collaborative DataWriters (Chapter 12 on page 727)
- Mechanisms for Achieving Information Durability and Persistence (Chapter 13 on page 732)
- Guaranteed Delivery of Data (Chapter 14 on page 752)
- Discovery (Chapter 15 on page 764)
- Transport Plugins (Chapter 16 on page 796)
- RTPS Locators and IP Mobility (Chapter 17 on page 827)
- Built-In Topics (Chapter 18 on page 831)
- Configuring QoS with XML (Chapter 19 on page 848)
- Multi-channel DataWriters (Chapter 20 on page 896)
- Connext DDS Threading Model (Chapter 21 on page 908)
- DDS Sample and Instance Memory Management (Chapter 22 on page 926)
- Sending Large Data (Chapter 23 on page 943)
- Topic Queries (Chapter 24 on page 977)
- Troubleshooting (Chapter 25 on page 981)
Chapter 11 Reliable Communications

Connext DDS uses best-effort delivery by default. The other type of delivery that Connext DDS supports is called reliable. This chapter provides instructions on how to set up and use reliable communication.

This chapter includes the following sections:

- 11.1 Sending Data Reliably below
- 11.2 Overview of the Reliable Protocol on page 690
- 11.3 Using QosPolicies to Tune the Reliable Protocol on page 694

11.1 Sending Data Reliably

The DCPS reliability model recognizes that the optimal balance between time-determinism and data-delivery reliability varies widely among applications and can vary among different publications within the same application. For example, individual DDS samples of signal data can often be dropped because their value disappears when the next DDS sample is sent. However, each DDS sample of command data must be received and it must be received in the order sent.

The QosPolicies provide a way to customize the determinism/reliability trade-off on a per Topic basis, or even on a per DataWriter/DataReader basis.

There are two delivery models:

- Best-effort delivery mode“I’m not concerned about missed or unordered DDS samples.”
- Reliable delivery model“Make sure all DDS samples get there, in order.”

11.1.1 Best-effort Delivery Model

By default, Connext DDS uses the best-effort delivery model: there is no effort spent ensuring in-order delivery or resending lost DDS samples. Best-effort DataReaders ignore lost DDS samples
in favor of the latest DDS sample. Your application is only notified if it does not receive a new DDS sample within a certain time period (set in the 7.5.7 DEADLINE QosPolicy on page 405).

The best-effort delivery model is best for time-critical information that is sent continuously. For instance, consider a DataWriter for the value of a sensor device (such as a the pressure inside a tank), and assume the DataWriter sends DDS samples continuously. In this situation, a DataReader for this Topic is only interested in having the latest pressure reading available—older DDS samples are obsolete.

### 11.1.2 Reliable Delivery Model

Reliable delivery means the DDS samples are guaranteed to arrive, in the order published.

The DataWriter maintains a send queue with space to hold the last $X$ number of DDS samples sent. Similarly, a DataReader maintains a receive queue with space for consecutive $X$ expected DDS samples.

The send and receive queues are used to temporarily cache DDS samples until Connext DDS is sure the DDS samples have been delivered and are not needed anymore. Connext DDS removes DDS samples from a publication’s send queue after the DDS sample has been acknowledged by all reliable subscriptions. When positive acknowledgements are disabled (see 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 and 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570), DDS samples are removed from the send queue after the corresponding keep-duration has elapsed (see Table 7.45 DDS_RtpsReliableWriterProtocol_t).

If an out-of-order DDS sample arrives, Connext DDS speculatively caches it in the DataReader’s receive queue (provided there is space in the queue). Only consecutive DDS samples are passed on to the DataReader.

DataWriters can be set up to wait for available queue space when sending DDS samples. This will cause the sending thread to block until there is space in the send queue. (Or, you can decide to sacrifice sending DDS samples reliably so that the sending rate is not compromised.) If the DataWriter is set up to ignore the full queue and sends anyway, then older cached DDS samples will be pushed out of the queue before all DataReaders have received them. In this case, the DataReader (or its Subscriber) is notified of the missing DDS samples through its Listener and/or Conditions.

Connext DDS automatically sends acknowledgments (ACKNACKs) as necessary to maintain reliable communications. The DataWriter may choose to block for a specified duration to wait for these acknowledgments (see 7.3.11 Waiting for Acknowledgments in a DataWriter on page 317).

Connext DDS establishes a virtual reliable channel between the matching DataWriter and all DataReaders. This mechanism isolates DataReaders from each other, allows the application to control memory usage, and provides mechanisms for the DataWriter to balance reliability and determinism. Moreover, the use of send and receive queues allows Connext DDS to be implemented efficiently without introducing unnecessary delays in the stream.

Note that a successful return code (DDS_RETCODE_OK) from write() does not necessarily mean that all DataReaders have received the data. It only means that the DDS sample has been added to the
DataWriter’s queue. To see if all DataReaders have received the data, look at the 7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305 to see if any DDS samples are unacknowledged.

Suppose DataWriter A reliably publishes a Topic to which DataReaders B and C reliably subscribe. B has space in its queue, but C does not. Will DataWriter A be notified? Will DataReader C receive any error messages or callbacks? The exact behavior depends on the QoS settings:

- If HISTORY_KEEP_ALL is specified for C, C will reject DDS samples that cannot be put into the queue and request A to resend missing DDS samples. The Listener is notified with the on_sample_rejected() callback (see 8.3.7.8 SAMPLE_REJECTED Status on page 531). If A has a queue large enough, or A is no longer writing new DDS samples, A won’t notice unless it checks the 7.3.6.8 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) on page 305.
- If HISTORY_KEEP_LAST is specified for C, C will drop old DDS samples and accept new ones. To A, it is as if all DDS samples have been received by C (that is, they have all been acknowledged).

### 11.2 Overview of the Reliable Protocol

An important advantage of Connext DDS is that it can offer the reliability and other QoS guarantees mandated by DDS on top of a very wide variety of transports, including packet-based transports, unreliable networks, multicast-capable transports, bursty or high-latency transports, etc. Connext DDS is also capable of maintaining liveliness and application-level QoS even in the presence of sporadic connectivity loss at the transport level, an important benefit in mobile networks. Connext DDS accomplishes this by implementing a reliable protocol that sequences and acknowledges application-level messages and monitors the liveliness of the link. This is called the Real-Time Publish-Subscribe (RTPS) protocol; it is an open, international standard.  

In order to work in this wide range of environments, the reliable protocol defined by RTPS is highly configurable with a set of parameters that let the application fine-tune its behavior to trade-off latency, responsiveness, liveness, throughput, and resource utilization. This section describes the most important features to the extent needed to understand how the configuration parameters affect its operation.

The most important features of the RTPS protocol are:

- Support for both push and pull operating modes
- Support for both positive and negative acknowledgments
- Support for high data-rate DataWriters

---

1For a link to the RTPS specification, see the RTI website, [www.rti.com](http://www.rti.com).
11.2 Overview of the Reliable Protocol

- Support for multicast DataReaders
- Support for high-latency environments

In order to support these features, RTPS uses several types of messages: Data messages (DATA), acknowledgments (ACKNACKs), and heartbeats (HBs).

- **DATA** messages contain snapshots of the value of data-objects and associate the snapshot with a sequence number that Connext DDS uses to identify them within the DataWriter’s history. These snapshots are stored in the history as a direct result of the application calling write() on the DataWriter. Incremental sequence numbers are automatically assigned by the DataWriter each time write() is called. In Figure 11.1: Basic RTPS Reliable Protocol on the next page through 11.3 Using QosPolicies to Tune the Reliable Protocol on page 694, these messages are represented using the notation DATA(<value>, <sequenceNum>). For example, DATA(A,1) represents a message that communicates the value ‘A’ and associates the sequence number ‘1’ with this message. A DATA is used for both keyed and non-keyed data types.

- **HB** messages announce to the DataReader that it should have received all snapshots up to the one tagged with a range of sequence numbers and can also request the DataReader to send an acknowledgement back. For example, HB(1-3) indicates to the DataReader that it should have received snapshots tagged with sequence numbers 1, 2, and 3 and asks the DataReader to confirm this.

- **ACKNACK** messages communicate to the DataWriter that particular snapshots have been successfully stored in the DataReader’s history. ACKNACKs also tell the DataWriter which snapshots are missing on the DataReader side. The ACKNACK message includes a set of sequence numbers represented as a bit map. The sequence numbers indicate which ones the DataReader is missing. (The bit map contains the base sequence number that has not been received, followed by the number of bits in bit map and the optional bit map. The maximum size of the bit map is 256.) All numbers up to (not including) those in the set are considered positively acknowledged. They are represented in Figure 11.1: Basic RTPS Reliable Protocol on the next page through Figure 11.7: Use of heartbeat_period on page 706 as ACKNACK(<first-missing>) or ACKNACK(<first-missing>-<last-missing>). For example, ACKNACK(4) indicates that the snapshots with sequence numbers 1, 2, and 3 have been successfully stored in the DataReader history, and that 4 has not been received.

It is important to note that Connext DDS can bundle multiple of the above messages within a single network packet. This ‘submessage bundling’ provides for higher performance communications.

It is also worth noting that because HB and ACKNACK messages communicate the state of reliable communication between individual writer and reader pairs, Connext DDS requires at least one unicast destination so that these messages can be sent to the correct destinations, as opposed to being broadcast over a multicast destination. Connext DDS does support enabling sending periodic heartbeats to a multicast destination using the enable_multicast_periodic_heartbeat in the 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670.
Figure 11.1: Basic RTPS Reliable Protocol

Figure 11.1: Basic RTPS Reliable Protocol above illustrates the basic behavior of the protocol when an application calls the write() operation on a DataWriter that is associated with a DataReader. As mentioned, the RTPS protocol can bundle multiple submessages into a single network packet. In Figure 11.1: Basic RTPS Reliable Protocol above this feature is used to piggyback a HB message to the DATA message. Note that before the message is sent, the data is given a sequence number (1 in this case) which is stored in the DataWriter’s send queue. As soon as the message is received by the DataReader, it places it into the DataReader’s receive queue. From the sequence number the DataReader can tell that it has not missed any messages and therefore it can make the data available immediately to the user (and call the DataReaderListener). This is indicated by the “✓” symbol. The reception of the HB(1) causes the DataReader to check that it has indeed received all updates up to and including the one with sequenceNumber=1. Since this is true, it replies with an ACKNACK(2) to positively acknowledge all messages up to (but not including) sequence number 2. The DataWriter notes that the update has been acknowledged, so it no longer needs to be retained in its send queue. This is indicated by the “✓” symbol.
Figure 11.2: RTPS Reliable Protocol in the Presence of Message Loss above illustrates the behavior of the protocol in the presence of lost messages. Assume that the message containing DATA(A,1) is dropped by the network. When the DataReader receives the next message (DATA(B,2); HB(1-2)) the DataReader will notice that the data associated with sequence number 1 was never received. It realizes this because the heartbeat HB(1-2) tells the DataReader that it should have received all messages up to and including the one with sequence number 2. This realization has two consequences:

- The data associated with sequence number 2 (B) is tagged with ‘X’ to indicate that it is not deliverable to the application (that is, it should not be made available to the application, because the
application needs to receive the data associated with DDS sample 1 (A) first).

- An ACKNACK(1) is sent to the DataWriter to request that the data tagged with sequence number 1 be resent.

Reception of the ACKNACK(1) causes the DataWriter to resend DATA(A,1). Once the DataReader receives it, it can ‘commit’ both A and B such that the application can now access both (indicated by the “✓”) and call the DataReaderListener. From there on, the protocol proceeds as before for the next data message (C) and so forth.

A subtle but important feature of the RTPS protocol is that ACKNACK messages are only sent as a direct response to HB messages. This allows the DataWriter to better control the overhead of these ‘administrative’ messages. For example, if the DataWriter knows that it is about to send a chain of DATA messages, it can bundle them all and include a single HB at the end, which minimizes ACKNACK traffic.

## 11.3 Using QosPolicies to Tune the Reliable Protocol

Reliability is controlled by the QosPolicies in Table 11.1 QosPolicies for Reliable Communications. To enable reliable delivery, read the following sections to learn how to change the QoS for the DataWriter and DataReader:

- 11.3.1 Enabling Reliability on page 696
- 11.3.2 Tuning Queue Sizes and Other Resource Limits on page 696
- 11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy on page 704
- 11.3.5 Avoiding Message Storms with DataReaderProtocol QosPolicy on page 712
- 11.3.6 Resending DDS Samples to Late-Joiners with the Durability QosPolicy on page 712

Then see 11.3.7 Use Cases on page 713 to explore example use cases:

### Table 11.1 QosPolicies for Reliable Communications

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
<th>Related Entities</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability</td>
<td>To establish reliable communication, this QoS must be set to DDS_RELIABLE_RELIABILITY_QOS for the DataWriter and its DataReaders.</td>
<td>DW, DR</td>
<td>11.3.1 Enabling Reliability on page 696, 7.5.21 RELIABILITY QosPolicy on page 447</td>
</tr>
</tbody>
</table>

\[DW = \text{DataWriter}, \ DR = \text{DataReader}\]
### Table 11.1 QosPolicies for Reliable Communications

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Description</th>
<th>Related Entities</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ResourceLimits</strong></td>
<td>This QoS determines the amount of resources each side can use to manage instances and DDS samples of instances. Therefore it controls the size of the DataWriter’s send queue and the DataReader’s receive queue. The send queue stores DDS samples until they have been ACKed by all DataReaders. The DataReader’s receive queue stores DDS samples for the user’s application.</td>
<td>DW, DR</td>
<td>11.3.2 Tuning Queue Sizes and Other Resource Limits on the next page, 7.5.22 RESOURCE_LIMI TE QosPolicy on page 451</td>
</tr>
<tr>
<td><strong>History</strong></td>
<td>This QoS affects how a DataWriter/DataReader behaves when its send/receive queue fills up.</td>
<td>DW, DR</td>
<td>11.3.3 Controlling Queue Depth with the History QosPolicy on page 703, 7.5.12 HISTORY QosPolicy on page 420</td>
</tr>
<tr>
<td><strong>DataWriterProtocol</strong></td>
<td>This QoS configures DataWriter-specific protocol. The QoS can disable positive ACKs for its DataReaders.</td>
<td>DW</td>
<td>11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy on page 704, 7.5.5 DATA_WRITER_PRO TOCOL QosPolicy (DDS Extension) on page 389</td>
</tr>
<tr>
<td><strong>DataReaderProtocol</strong></td>
<td>When a reliable DataReader receives a heartbeat from a DataWriter and needs to return an ACKNACK, the DataReader can choose to delay a while. This QoS sets the minimum and maximum delay. It can also disable positive ACKs for the DataReader.</td>
<td>DR</td>
<td>11.3.5 Avoiding Message Storms with DataReaderProtocol QosPolicy on page 712, 7.5.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570</td>
</tr>
<tr>
<td><strong>DataReaderResourceLimits</strong></td>
<td>This QoS determines additional amounts of resources that the DataReader can use to manage DDS samples (namely, the size of the DataReader’s internal queues, which cache DDS samples until they are ordered for reliability and can be moved to the DataReader’s receive queue for access by the user’s application).</td>
<td>DR</td>
<td>11.3.6 Resending DDS Samples to Late-Joiners with the Durability QosPolicy on page 712, 7.5.9 DURABILITY QosPolicy on page 411</td>
</tr>
<tr>
<td><strong>Durability</strong></td>
<td>This QoS affects whether late-joining DataReaders will receive all previously-sent data or not.</td>
<td>DW, DR</td>
<td>11.3.6 Resending DDS Samples to Late-Joiners with the Durability QosPolicy on page 712, 7.5.9 DURABILITY QosPolicy on page 411</td>
</tr>
</tbody>
</table>

1DW = DataWriter, DR = DataReader
11.3.1 Enabling Reliability

You must modify the 7.5.21 RELIABILITY QosPolicy on page 447 of the DataWriter and each of its reliable DataReaders. Set the kind field to DDS_RELIABLE_RELIABILITY_QOS:

- DataWriter
  
  ```
  writer_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
  ```

- DataReader
  
  ```
  reader_qos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
  ```

11.3.1.1 Blocking until the Send Queue Has Space Available

The max_blocking_time property in the 7.5.21 RELIABILITY QosPolicy on page 447 indicates how long a DataWriter can be blocked during a write().

If max_blocking_time is non-zero and the reliability send queue is full, the write is blocked (the DDS sample is not sent). If max_blocking_time has passed and the DDS sample is still not sent, write() returns DDS_RETCODE_TIMEOUT and the DDS sample is not sent.

If the number of unacknowledged DDS samples in the reliability send queue drops below max_samples (set in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451) before max_blocking_time, the DDS sample is sent and write() returns DDS_RETCODE_OK.

If max_blocking_time is zero and the reliability send queue is full, write() returns DDS_RETCODE_TIMEOUT and the DDS sample is not sent.

11.3.2 Tuning Queue Sizes and Other Resource Limits

Set the 7.5.12 HISTORY QosPolicy on page 420 appropriately to accommodate however many DDS samples should be saved in the DataWriter’s send queue or in the DataReader’s receive queue.

Set the size of the send window in the DDS_RtpsReliableWriterProtocol_t policy (in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389) appropriately to accommodate the maximum number of unacknowledged DDS samples that can be queued at a time from a DataWriter.

For more information, see the following sections:

- 11.3.2.1 Understanding the Send Queue and Setting its Size on the next page
- 11.3.2.2 Understanding the Receive Queue and Setting Its Size on page 700
- 7.5.5.4 Configuring the Send Window Size on page 397

Note: The HistoryQosPolicy’s depth must be less than or equal to the ResourceLimitsQosPolicy’s max_samples_per_instance; max_samples_per_instance must be less than or equal to the
11.3.2 Tuning Queue Sizes and Other Resource Limits

ResourceLimitsQosPolicy’s **max_samples** (see 7.5.22 RESOURCE_LIMITS QosPolicy on page 451), and **max_samples_per_remote_writer** (see 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576) must be less than or equal to **max_samples**.

- depth <= max_samples_per_instance <= max_samples
- max_samples_per_remote_writer <= max_samples

**Examples:**

**DataWriter**

```plaintext
writer_qos.resource_limits.initial_instances = 10;
writer_qos.resource_limits.initial_samples = 200;
writer_qos.resource_limits.max_instances = 100;
writer_qos.resource_limits.max_samples = 2000;
writer_qos.resource_limits.max_samples_per_instance = 20;
writer_qos.history.depth = 20;
```

**DataReader**

```plaintext
reader_qos.resource_limits.initial_instances = 10;
reader_qos.resource_limits.initial_samples = 200;
reader_qos.resource_limits.max_instances = 100;
reader_qos.resource_limits.max_samples = 2000;
reader_qos.resource_limits.max_samples_per_instance = 20;
reader_qos.history.depth = 20;
reader_qos.reader_resource_limits.max_samples_per_remote_writer = 20;
```

11.3.2.1 Understanding the Send Queue and Setting its Size

A **DataWriter**’s send queue is used to store each DDS sample it writes. A DDS sample will be removed from the send queue after it has been acknowledged (through an ACKNACK) by all the reliable **DataReaders**. A **DataReader** can request that the **DataWriter** resend a missing DDS sample (through an ACKNACK). If that DDS sample is still available in the send queue, it will be resent. To elicit timely ACKNACKs, the **DataWriter** will regularly send heartbeats to its reliable **DataReaders**.

A **DataWriter**’s send queue size is determined by its 7.5.22 RESOURCE_LIMITS QosPolicy on page 451, specifically the **max_samples** field. The appropriate value depends on application parameters such as how fast the publication calls **write()**.

A **DataWriter** has a "send window" that is the maximum number of unacknowledged DDS samples allowed in the send queue before a **DataWriter** will start blocking during the **write()** call (see 7.3.8.1 Blocking During a write() on page 312). The send window enables throttling of the publishing application to avoid overwhelming matched **DataReaders**. If the **DataReaders** are not acknowledging samples fast enough and the **DataWriter**’s send window fills up, the **DataWriter** will be slowed down because each **write()** call will block until the unacknowledged sample count in the send window decreases.
The size of the send window is determined by the DataWriterProtocolQosPolicy, specifically the fields min_send_window_size and max_send_window_size within the rtps_reliable_writer field of type DDS_RtpsReliableWriterProtocol_t. Other fields can be used to configure a variable-sized send window, where the send window size changes in response to network congestion to maximize the effective send rate. Like for max_samples, the appropriate values depend on application parameters. For more information on configuring the send window size, refer to 7.5.5.4 Configuring the Send Window Size on page 397.

Strict reliability: If a DataWriter does not receive ACKNACKs from one or more reliable DataReaders, it is possible for the reliability send queue—either its finite max_send_window_size or its effective max_send_window_size if max_send_window_size is infinite—to fill up. Effective max_send_window_size is defined as either max_samples (if batching is not used) or max_batches (if batching is used). If you want to achieve strict reliability, the kind field in the 7.5.12 HISTORY QosPolicy on page 420 for both the DataReader and DataWriter must be set to KEEP_ALL, positive acknowledgments must be enabled for both the DataReader and DataWriter, and your publishing application should wait until space is available in the reliability queue before writing any more DDS samples. Connext DDS provides two mechanisms to do this:

- Allow the write() operation to block until there is space in the reliability queue again to store the DDS sample. The maximum time this call blocks is determined by the max_blocking_time field in the 7.5.21 RELIABILITY QosPolicy on page 447 (also discussed in 11.3.1.1 Blocking until the Send Queue Has Space Available on page 696).
- Use the DataWriter’s Listener to be notified when the reliability queue fills up or empties again.

When the 7.5.12 HISTORY QosPolicy on page 420 on the DataWriter is set to KEEP_LAST, strict reliability is not guaranteed. When there are depth number of DDS samples in the queue (set in the 7.5.12 HISTORY QosPolicy on page 420, see 11.3.3 Controlling Queue Depth with the History QosPolicy on page 703) the oldest DDS sample will be dropped from the queue when a new DDS sample is written. Note that in such a reliable mode, when the send window is larger than max_samples (or max_batches if batching is enabled), the DataWriter will never block, but strict reliability is no longer guaranteed. If there is a request for the purged DDS sample from any DataReaders, the DataWriter will send a heartbeat that no longer contains the sequence number of the dropped DDS sample (it will not be able to send the DDS sample).

Alternatively, a DataWriter with KEEP_LAST may block on write() when its send window is smaller than its send queue. The DataWriter will block when its send window is full. After the blocking time has elapsed, the DataWriter may replace a DDS sample, regardless of its acknowledgement status. See 7.3.8.2 write() behavior with KEEP_LAST and KEEP_ALL on page 312 for a detailed explanation of what happens when certain limits are reached during a call to write().

The send queue size is set in the max_samples field of the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451. The appropriate size for the send queue depends on application parameters (such as the send...
11.3.2 Tuning Queue Sizes and Other Resource Limits

rate), channel parameters (such as end-to-end delay and probability of packet loss), and quality of service requirements (such as maximum acceptable probability of DDS sample loss).

The DataReader’s receive queue size should generally be larger than the DataWriter’s send queue size. Receive queue size is discussed in 11.3.2.2 Understanding the Receive Queue and Setting Its Size on the next page.

A good rule of thumb, based on a simple model that assumes individual packet drops are not correlated and time-independent, is that the size of the reliability send queue, N, is as shown in Figure 11.3: Calculating Minimum Send Queue Size for a Desired Level of Reliability below.

**Figure 11.3: Calculating Minimum Send Queue Size for a Desired Level of Reliability**

\[ N = 2RT(\log(1-Q))/\log(p) \]

*Simple formula for determining the minimum size of the send queue required for strict reliability*

In the above equation, \( R \) is the rate of sending DDS samples, \( T \) is the round-trip transmission time, \( p \) is the probability of a packet loss in a round trip, and \( Q \) is the required probability that a DDS sample is eventually successfully delivered. Of course, network-transport dropouts must also be taken into account and may influence or dominate this calculation.

Table 11.2 Required Size of the Send Queue for Different Network Parameters gives the required size of the send queue for several common scenarios.

<table>
<thead>
<tr>
<th>Q(^1)</th>
<th>p(^2)</th>
<th>T(^3)</th>
<th>R(^4)</th>
<th>N(^5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>99%</td>
<td>1%</td>
<td>0.001(^6) sec</td>
<td>100 Hz</td>
<td>1</td>
</tr>
<tr>
<td>99%</td>
<td>1%</td>
<td>0.001 sec</td>
<td>2000 Hz</td>
<td>2</td>
</tr>
<tr>
<td>99%</td>
<td>5%</td>
<td>0.001 sec</td>
<td>100 Hz</td>
<td>1</td>
</tr>
<tr>
<td>99%</td>
<td>5%</td>
<td>0.001 sec</td>
<td>2000 Hz</td>
<td>4</td>
</tr>
</tbody>
</table>

\(^1\)Q\(^1\) is the desired level of reliability measured as the probability that any data update will eventually be delivered successfully. In other words, percentage of DDS samples that will be successfully delivered.

\(^2\)p\(^2\) is the probability that any single packet gets lost in the network.

\(^3\)T\(^3\) is the round-trip transport delay in the network.

\(^4\)R\(^4\) is the rate at which the publisher is sending updates.

\(^5\)N\(^5\) is the minimum required size of the send queue to accomplish the desired level of reliability "Q".

\(^6\)The typical round-trip delay for a dedicated 100 Mbit/second ethernet is about 0.001 seconds.
11.3.2 Tuning Queue Sizes and Other Resource Limits

Table 11.2 Required Size of the Send Queue for Different Network Parameters

<table>
<thead>
<tr>
<th>Q¹</th>
<th>p²</th>
<th>T³</th>
<th>R⁴</th>
<th>N⁵</th>
</tr>
</thead>
<tbody>
<tr>
<td>99.99%</td>
<td>1%</td>
<td>0.001 sec</td>
<td>100 Hz</td>
<td>1</td>
</tr>
<tr>
<td>99.99%</td>
<td>1%</td>
<td>0.001 sec</td>
<td>2000 Hz</td>
<td>6</td>
</tr>
<tr>
<td>99.99%</td>
<td>5%</td>
<td>0.001 sec</td>
<td>100 Hz</td>
<td>1</td>
</tr>
<tr>
<td>99.99%</td>
<td>5%</td>
<td>0.001 sec</td>
<td>2000 Hz</td>
<td>8</td>
</tr>
</tbody>
</table>

Note: Packet loss on a network frequently happens in bursts, and the packet loss events are correlated. This means that the probability of a packet being lost is much higher if the previous packet was lost because it indicates a congested network or busy receiver. For this situation, it may be better to use a queue size that can accommodate the longest period of network congestion, as illustrated in Figure 11.4: Calculating Minimum Send Queue Size for Networks with Dropouts below.

Figure 11.4: Calculating Minimum Send Queue Size for Networks with Dropouts

\[ N = RD(Q) \]

Send queue size as a function of send rate "R" and maximum dropout time D

In the above equation R is the rate of sending DDS samples, D(Q) is a time such that Q percent of the dropouts are of equal or lesser length, and Q is the required probability that a DDS sample is eventually successfully delivered. The problem with the above formula is that it is hard to determine the value of D(Q) for different values of Q.

For example, if we want to ensure that 99.9% of the DDS samples are eventually delivered successfully, and we know that the 99.9% of the network dropouts are shorter than 0.1 seconds, then we would use \( N = 0.1*R \). So for a rate of 100Hz, we would use a send queue of \( N = 10 \); for a rate of 2000Hz, we would use \( N = 200 \).

11.3.2.2 Understanding the Receive Queue and Setting Its Size

DDS samples are stored in the DataReader’s receive queue, which is accessible to the user’s application.

¹\(^\text{"Q"}\) is the desired level of reliability measured as the probability that any data update will eventually be delivered successfully. In other words, percentage of DDS samples that will be successfully delivered.

²\(^\text{"p"}\) is the probability that any single packet gets lost in the network.

³\(^\text{"T"}\) is the round-trip transport delay in the network

⁴\(^\text{"R"}\) is the rate at which the publisher is sending updates.

⁵\(^\text{"N"}\) is the minimum required size of the send queue to accomplish the desired level of reliability "Q".
A DDS sample is removed from the receive queue after it has been accessed by take(), as described in 8.4.3 Accessing DDS Data Samples with Read or Take on page 555. Note that read() does not remove DDS samples from the queue.

A DataReader’s receive queue size is limited by its 7.5.22 RESOURCE_LIMITS QosPolicy on page 451, specifically the max_samples field. The storage of out-of-order DDS samples for each DataWriter is also allocated from the DataReader’s receive queue; this DDS sample resource is shared among all reliable DataWriters. That is, max_samples includes both ordered and out-of-order DDS samples.

A DataReader can maintain reliable communications with multiple DataWriters (e.g., in the case of the 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438 setting of SHARED). The maximum number of out-of-order DDS samples from any one DataWriter that can occupy in the receive queue is set in the max_samples_per_remote_writer field of the 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576; this value can be used to prevent a single DataWriter from using all the space in the receive queue. max_samples_per_remote_writer must be set to be <= max_samples.

The DataReader will cache DDS samples that arrive out of order while waiting for missing DDS samples to be resent. (Up to 256 DDS samples can be resent; this limitation is imposed by the wire protocol.) If there is no room, the DataReader has to reject out-of-order DDS samples and request them again later after the missing DDS samples have arrived.

The appropriate size of the receive queue depends on application parameters, such as the DataWriter’s sending rate and the probability of a dropped DDS sample. However, the receive queue size should generally be larger than the send queue size. Send queue size is discussed in 11.3.2.1 Understanding the Send Queue and Setting its Size on page 697.

Figure 11.5: Effect of Receive-Queue Size on Performance: Large Queue Size on the next page and Figure 11.6: Effect of Receive Queue Size on Performance: Small Queue Size on page 703 compare two hypothetical DataReaders, both interacting with the same DataWriter. The queue on the left represents an ordering cache, allocated from receive queue—DDS samples are held here if they arrive out of order. The DataReader in Figure 11.5: Effect of Receive-Queue Size on Performance: Large Queue Size on the next page has a sufficiently large receive queue (max_samples) for the given send rate of the DataWriter and other operational parameters. In both cases, we assume that all DDS samples are taken from the DataReader in the Listener callback. (See 8.4.3 Accessing DDS Data Samples with Read or Take on page 555 for information on take() and related operations.)

In Figure 11.6: Effect of Receive Queue Size on Performance: Small Queue Size on page 703, max_samples is too small to cache out-of-order DDS samples for the same operational parameters. In both cases, the DataReaders eventually receive all the DDS samples in order. However, the DataReader with the larger max_samples will get the DDS samples earlier and with fewer transactions. In particular, DDS sample “4” is never resent for the DataReader with the larger queue size.
Figure 11.5: Effect of Receive-Queue Size on Performance: Large Queue Size

- Send sample 1
- Send sample 2
  - Sample 2 lost
- Send sample 3
  - Send Heartbeat
- Send sample 4
- Resend sample 2
  - ACKNACK
- Send sample 5

DataWriter

DataReader

max_samples is 4. This also limits
the number of unordered samples
that can be cached.

1

Sample 1 is taken.

Note: No unordered samples cached.

3

Space reserved for missing sample 2.

3 4

Samples 3 and 4 are cached while
waiting for sample 2.

2 3 4

Samples 2-4 are taken.

5

Sample 5 is taken.
11.3.3 Controlling Queue Depth with the History QosPolicy

If you want to achieve strict reliability, set the `kind` field in the 7.5.12 HISTORY QosPolicy on page 420 for both the `DataReader` and `DataWriter` to `KEEP_ALL`; in this case, the `depth` does not matter.

Or, for non-strict reliability, you can leave the `kind` set to `KEEP_LAST` (the default). This will provide non-strict reliability; some DDS samples may not be delivered if the resource limit is reached.

The `depth` field in the 7.5.12 HISTORY QosPolicy on page 420 controls how many DDS samples `Connext DDS` will attempt to keep on the `DataWriter`’s send queue or the `DataReader`’s receive queue. For reliable communications, depth should be $>= 1$. The depth can be set to 1, but cannot be more than the `max_samples_per_instance` in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.
11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy

Example:

- **DataWriter**

```plaintext
datawriter_qos.history.depth = <number of DDS samples to keep in send queue>;
```

- **DataReader**

```plaintext
datareader_qos.history.depth = <number of DDS samples to keep in receive queue>;
```

11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy

In the Connext DDS reliability model, the DataWriter sends DDS data samples and heartbeats to reliable DataReaders. A DataReader responds to a heartbeat by sending an ACKNACK, which tells the DataWriter what the DataReader has received so far.

In addition, the DataReader can request missing DDS samples (by sending an ACKNACK) and the DataWriter will respond by resending the missing DDS samples. This section describes some advanced timing parameters that control the behavior of this mechanism. Many applications do not need to change these settings. These parameters are contained in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

The protocol described in 11.2 Overview of the Reliable Protocol on page 690 uses very simple rules such as piggybacking HB messages to each DATA message and responding immediately to ACKNACKs with the requested repair messages. While correct, this protocol would not be capable of accommodating optimum performance in more advanced use cases.

This section describes some of the parameters configurable by means of the rtps_reliable_writer structure in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 and how they affect the behavior of the RTPS protocol.

11.3.4.1 How Often Heartbeats are Resent (heartbeat_period)

If a DataReader does not acknowledge a DDS sample that has been sent, the DataWriter resends the heartbeat. These heartbeats are resent at the rate set in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389, specifically its heartbeat_period field.

For example, a heartbeat_period of 3 seconds means that if a DataReader does not receive the latest DDS sample (for example, it gets dropped by the network), it might take up to 3 seconds before the DataReader realizes it is missing data. The application can lower this value when it is important that recovery from packet loss is very fast.

The basic approach of sending HB messages as a piggyback to DATA messages has the advantage of minimizing network traffic. However, there is a situation where this approach, by itself, may result in large latencies. Suppose there is a DataWriter that writes bursts of data, separated by relatively long periods of silence. Furthermore assume that the last message in one of the bursts is lost by the network. This is the
11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy

case shown for message DATA(B, 2) in Figure 11.7: Use of heartbeat_period on the next page. If HBs were only sent piggybacked to DATA messages, the DataReader would not realize it missed the ‘B’ DATA message with sequence number ‘2’ until the DataWriter wrote the next message. This may be a long time if data is written sporadically. To avoid this situation, Connext DDS can be configured so that HBs are sent periodically as long as there are DDS samples that have not been acknowledged even if no data is being sent. The period at which these HBs are sent is configurable by setting the rtps_reliable_writer.heartbeat_period field in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

Note that a small value for the heartbeat_period will result in a small worst-case latency if the last message in a burst is lost. This comes at the expense of the higher overhead introduced by more frequent HB messages.

Also note that the heartbeat_period should not be less than the rtps_reliable_reader.heartbeat_suppression_duration in the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570; otherwise those HBs will be lost.
11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy

11.3.4.2 How Often Piggyback Heartbeats are Sent (heartbeats_per_max_samples)

A DataWriter will automatically send heartbeats with new DDS samples to request regular ACKNACKs from the DataReader. These are called “piggyback” heartbeats.

A piggyback heartbeat is sent every \( \left( \frac{\text{current send-window size}}{\text{heartbeats_per_max_samples}} \right) \) number of DDS samples written.

The heartbeats_per_max_samples field is part of the rtps_reliable_writer structure in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389. If heartbeats_per_max_
samples is set equal to max_send_window_size, this means that a heartbeat will be sent with each DDS sample. A value of 8 means that a heartbeat will be sent with every 'current send-window size/8' DDS samples. Say current send window is 1024, then a heartbeat will be sent once every 128 DDS samples. If you set this to zero, DDS samples are sent without any piggyback heartbeat. The max_send_window_size field is part of the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

Figure 11.1: Basic RTPS Reliable Protocol and Figure 11.2: RTPS Reliable Protocol in the Presence of Message Loss seem to imply that a heartbeat (HB) is sent as a piggyback to each DATA message. However, in situations where data is sent continuously at high rates, piggybacking a HB to each message may result in too much overhead; not so much on the HB itself, but on the ACKNACKs that would be sent back as replies by the DataReader.

There are two reasons to send a HB:

- To request that a DataReader confirm the receipt of data via an ACKNACK, so that the DataWriter can remove it from its send queue and therefore prevent the DataWriter’s history from filling up (which could cause the write() operation to temporarily block\(^1\)).
- To inform the DataReader of what data it should have received, so that the DataReader can send a request for missing data via an ACKNACK.

The DataWriter’s send queue can buffer many DDS data samples while it waits for ACKNACKs, and the DataReader’s receive queue can store out-of-order DDS samples while it waits for missing ones. So it is possible to send HB messages much less frequently than DATA messages. The ratio of piggyback HB messages to DATA messages is controlled by the rtps_reliable_writer.heartbeats_per_max_samples field in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

A HB is used to get confirmation from DataReaders so that the DataWriter can remove acknowledged DDS samples from the queue to make space for new DDS samples. Therefore, if the queue size is large, or new DDS samples are added slowly, HBs can be sent less frequently.

In Figure 11.8: Use of heartbeats_per_max_samples on the next page, the DataWriter sets the heartbeats_per_max_samples to certain value so that a piggyback HB will be sent for every three DDS samples. The DataWriter first writes DDS sample A and B. The DataReader receives both. However, since no HB has been received, the DataReader won’t send back an ACKNACK. The DataWriter will still keep all the DDS samples in its queue. When the DataWriter sends DDS sample C, it will send a piggyback HB along with the DDS sample. Once the DataReader receives the HB, it will send back an ACKNACK for DDS samples up to sequence number 3, such that the DataWriter can remove all three DDS samples from its queue.

\(^1\)Note that data could also be removed from the DataWriter’s send queue if it is no longer relevant due to some other QoS such a HISTORY KEEP_LAST (7.5.12 HISTORY QosPolicy on page 420) or LIFESPAN (7.5.14 LIFESPAN QoS Policy on page 425).
11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy

### 11.3.4.3 Controlling Packet Size for Resent DDS Samples (max_bytes_per_nack_response)

A DataWriter may resend multiple missed DDS samples in the same packet. The `max_bytes_per_nack_response` field in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 limits the size of this ‘repair’ packet. The reliable DataWriter will include at least one sample in the repair packet.

For example, if the DataReader requests 20 DDS samples, each 10K, and the `max_bytes_per_nack_response` is set to 100K, the DataWriter will only send the first 10 DDS samples at most. The DataReader will have to ACKNACK again to receive the other DDS samples.
Regardless of this setting, the maximum number of samples that can be part of a repair packet is limited to 32. This limit cannot be changed by configuration. In addition, the number of samples is limited by the value of NDDS_Transport_Property_t’s gather_send_buffer_count_max (see 16.6.1 Setting the Maximum Gather-Send Buffer Count for UDP Transports on page 819).

11.3.4.4 Controlling How Many Times Heartbeats are Resent (max_heartbeat_retries)

If a DataReader does not respond within max_heartbeat_retries number of heartbeats, it will be dropped by the DataWriter and the reliable DataWriter’s Listener will be called with a 7.3.6.9 RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) on page 307.

If the dropped DataReader becomes available again (perhaps its network connection was down temporarily), it will be added back to the DataWriter the next time the DataWriter receives some message (ACKNACK) from the DataReader.

When a DataReader is ‘dropped’ by a DataWriter, the DataWriter will not wait for the DataReader to send an ACKNACK before any DDS samples are removed. However, the DataWriter will still send data and HBs to this DataReader as normal.

The max_heartbeat_retries field is part of the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

11.3.4.5 Treating Non-Progressing Readers as Inactive Readers (inactivate_nonprogressing_readers)

In addition to max_heartbeat_retries, if inactivate_nonprogressing_readers is set, then not only are non-responsive DataReaders considered inactive, but DataReaders sending non-progressing NACKs can also be considered inactive. A non-progressing NACK is one which requests the same oldest DDS sample as the previously received NACK. In this case, the DataWriter will not consider a non-progressing NACK as coming from an active reader, and hence will inactivate the DataReader if no new NACKs are received before max_heartbeat_retries number of heartbeat periods has passed.

One example for which it could be useful to turn on inactivate_nonprogressing_readers is when a DataReader’s (keep-all) queue is full of untaken historical DDS samples. Each subsequent heartbeat would trigger the same NACK, and nominally the DataReader would not be inactivated. A user not requiring strict-reliability could consider setting inactivate_nonprogressing_readers to allow the DataWriter to progress rather than being held up by this non-progressing DataReader.

11.3.4.6 Coping with Redundant Requests for Missing DDS Samples (max_nack_response_delay)

When a DataWriter receives a request for missing DDS samples from a DataReader and responds by resending the requested DDS samples, it will ignore additional requests for the same DDS samples during the time period max_nack_response_delay.
11.3.4 Controlling Heartbeats and Retries with DataWriterProtocol QosPolicy

The rtps_reliable_writer.max_nack_response_delay field is part of the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

If your send period is smaller than the round-trip delay of a message, this can cause unnecessary DDS sample retransmissions due to redundant ACKNACKs. In this situation, an ACKNACK triggered by an out-of-order DDS sample is not received before the next DDS sample is sent. When a DataReader receives the next message, it will send another ACKNACK for the missing DDS sample. As illustrated in Figure 11.9: Resending Missing Samples due to Duplicate ACKNACKs below, duplicate ACKNACK messages cause another resending of missing DDS sample “2” and lead to wasted CPU usage on both the publication and the subscription sides.

**Figure 11.9: Resending Missing Samples due to Duplicate ACKNACKs**

![Diagram showing resending of missing samples due to duplicate ACKNACKs](image)

While these redundant messages provide an extra cushion for the level of reliability desired, you can conserve the CPU and network bandwidth usage by limiting how often the same ACKNACK messages are sent; this is controlled by min_nack_response_delay.

Reliable subscriptions are prevented from resending an ACKNACK within min_nack_response_delay seconds from the last time an ACKNACK was sent for the same DDS sample. Our testing shows that the
default min_nack_response_delay of 0 seconds achieves an optimal balance for most applications on typical Ethernet LANs.

However, if your system has very slow computers and/or a slow network, you may want to consider increasing min_nack_response_delay. Sending an ACKNACK and resending a missing DDS sample inherently takes a long time in this system. So you should allow a longer time for recovery of the lost DDS sample before sending another ACKNACK. In this situation, you should increase min_nack_response_delay.

If your system consists of a fast network or computers, and the receive queue size is very small, then you should keep min_nack_response_delay very small (such as the default value of 0). If the queue size is small, recovering a missing DDS sample is more important than conserving CPU and network bandwidth (new DDS samples that are too far ahead of the missing DDS sample are thrown away). A fast system can cope with a smaller min_nack_response_delay value, and the reliable DDS sample stream can normalize more quickly.

11.3.4.7 Disabling Positive Acknowledgements (disable_positive_acks_min_sample_keep_duration)

When ACKNACK storms are a primary concern in a system, an alternative to tuning heartbeat and ACKNACK response delays is to disable positive acknowledgments (ACKs) and rely just on NACKs to maintain reliability. Systems with non-strict reliability requirements can disable ACKs to reduce network traffic and directly solve the problem of ACK storms. ACKs can be disabled for the DataWriter and the DataReader; when disabled for the DataWriter, none of its DataReaders will send ACKs, whereas disabling it at the DataReader allows per-DataReader configuration.

Normally when ACKs are enabled, strict reliability is maintained by the DataWriter, guaranteeing that a DDS sample stays in its send queue until all DataReaders have positively acknowledged it (aside from relevant DURABILITY, HISTORY, and LIFESPAN QoS policies). When ACKs are disabled, strict reliability is no longer guaranteed, but the DataWriter should still keep the DDS sample for a sufficient duration for ACK-disabled DataReaders to have a chance to NACK it. Thus, a configurable “keep-duration” (disable_positive_acks_min_sample_keep_duration) applies for DDS samples written for ACK-disabled DataReaders, where DDS samples are kept in the queue for at least that keep-duration. After the keep-duration has elapsed for a DDS sample, the DDS sample is considered to be “acknowledged” by its ACK-disabled DataReaders.

The keep duration should be configured for the expected worst-case from when the DDS sample is written to when a NACK for the DDS sample could be received. If set too short, the DDS sample may no longer be queued when a NACK requests it, which is the cost of not enforcing strict reliability.

If the peak send rate is known and writer resources are available, the writer queue can be sized so that writes will not block. For this case, the queue size must be greater than the send rate multiplied by the keep duration.
11.3.5 Avoiding Message Storms with DataReaderProtocol QosPolicy

DataWriters send DDS data samples and heartbeats to DataReaders. A DataReader responds to a heartbeat by sending an acknowledgement that tells the DataWriter what the DataReader has received so far and what it is missing. If there are many DataReaders, all sending ACKNACKs to the same DataWriter at the same time, a message storm can result. To prevent this, you can set a delay for each DataReader, so they don’t all send ACKNACKs at the same time. This delay is set in the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570.

If you have several DataReaders per DataWriter, varying this delay for each one can avoid ACKNACK message storms to the DataWriter. If you are not concerned about message storms, you do not need to change this QosPolicy.

Example:

```plaintext
reader_qos.protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
reader_qos.protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;
reader_qos.protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
reader_qos.protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec = 0.5 * 1000000000UL; // 0.5 sec
```

As the name suggests, the minimum and maximum response delay bounds the random wait time before the response. Setting both to zero will force immediate response, which may be necessary for the fastest recovery in case of lost DDS samples.

11.3.6 Resending DDS Samples to Late-Joiners with the Durability QosPolicy

The 7.5.9 DURABILITY QosPolicy on page 411 is also somewhat related to Reliability. Connext DDS requires a finite time to "discover" or match DataReaders to DataWriters. If an application attempts to send data before the DataReader and DataWriter "discover" one another, then the DDS sample will not actually get sent. Whether or not DDS samples are resent when the DataReader and DataWriter eventually "discover" one another depends on how the DURABILITY and 7.5.12 HISTORY QosPolicy on page 420 are set. The default setting for the Durability QosPolicy is VOLATILE, which means that the DataWriter will not store DDS samples for redelivery to late-joining DataReaders.

Connext DDS also supports the TRANSIENT_LOCAL setting for the Durability, which means that the DDS samples will be kept stored for redelivery to late-joining DataReaders, as long as the DataWriter is around. The DDS samples are not stored beyond the lifecycle of the DataWriter.

How many samples are sent to late-joining DataReaders is determined by the writer_depth in the DURABILITY QosPolicy.

See also: 8.3.6 Waiting for Historical Data on page 515.
11.3.7 Use Cases

This section contains advanced material that discusses practical applications of the reliability related QoS.

11.3.7.1 Importance of Relative Thread Priorities

For high throughput, the Connext DDS Event thread’s priority must be sufficiently high on the sending application. Unlike an unreliable writer, a reliable writer relies on internal Connext DDS threads: the Receive thread processes ACKNACKs from the DataReaders, and the Event thread schedules the events necessary to maintain reliable data flow.

- When DDS samples are sent to the same or another application on the same host, the Receive thread priority should be higher than the writing thread priority (priority of the thread calling write() on the DataWriter). This will allow the Receive thread to process the messages as they are sent by the writing thread. A sustained reliable flow requires the reader to be able to process the DDS samples from the writer at a speed equal to or faster than the writer emits.

- The default Event thread priority is low. This is adequate if your reliable transfer is not sustained; queued up events will eventually be processed when the writing thread yields the CPU. The Connext DDS can automatically grow the event queue to store all pending events. But if the reliable communication is sustained, reliable events will continue to be scheduled, and the event queue will eventually reach its limit. The default Event thread priority is unsuitable for maintaining a fast and sustained reliable communication and should be increased through the participant_qos.event.-thread.priority. This value maps directly to the OS thread priority, see 9.5.5 EVENT QosPolicy (DDS Extension) on page 662).

The Event thread should also be increased to minimize the reliable latency. If events are processed at a higher priority, dropped packets will be resent sooner.

Now we consider some practical applications of the reliability related QoS:

- 11.3.7.2 Aperiodic Use Case: One-at-a-Time below
- 11.3.7.3 Aperiodic, Bursty on page 718
- 11.3.7.4 Periodic on page 721

11.3.7.2 Aperiodic Use Case: One-at-a-Time

Suppose you have aperiodically generated data that needs to be delivered reliably, with minimum latency, such as a series of commands (“Ready,” “Aim,” “Fire”). If a writing thread may block between each DDS sample to guarantee reception of the just-sent DDS sample on the reader’s middleware end, a smaller queue will provide a smaller upper bound on the DDS sample delivery time. Adequate writer QoS for this use case are presented in Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer on the next page.
Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td><code>qos-&gt;reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;</code></td>
</tr>
<tr>
<td>2.</td>
<td><code>qos-&gt;history.kind = DDS_KEEP_ALL_HISTORY_QOS;</code></td>
</tr>
<tr>
<td>3.</td>
<td><code>qos-&gt;protocol.push_on_write = DDS_BOOLEAN_TRUE;</code></td>
</tr>
<tr>
<td>4.</td>
<td>//use these hard coded value unless you use a key</td>
</tr>
<tr>
<td>5.</td>
<td><code>qos-&gt;resource_limits.initial_samples = qos-&gt;resource_limits.max_samples = 1;</code></td>
</tr>
<tr>
<td>6.</td>
<td><code>qos-&gt;resource_limits.max_samples_per_instance =</code></td>
</tr>
<tr>
<td>7.</td>
<td><code>qos-&gt;resource_limits.max_samples;</code></td>
</tr>
<tr>
<td>8.</td>
<td><code>qos-&gt;resource_limits.max_instances =</code></td>
</tr>
<tr>
<td>9.</td>
<td><code>qos-&gt;resource_limits.max_instances = 1;</code></td>
</tr>
<tr>
<td>10.</td>
<td><code>qos-&gt;resource_limits.max_samples = 1;</code></td>
</tr>
<tr>
<td>11.</td>
<td><code>// want to piggyback HB w/ every sample.</code></td>
</tr>
<tr>
<td>12.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.heartbeats_per_max_samples =</code></td>
</tr>
<tr>
<td>13.</td>
<td><code>qos-&gt;resource_limits.max_samples;</code></td>
</tr>
<tr>
<td>14.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.high_watermark = 1;</code></td>
</tr>
<tr>
<td>15.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.low_watermark = 0;</code></td>
</tr>
<tr>
<td>16.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.min_nack_response_delay.sec = 0;</code></td>
</tr>
<tr>
<td>17.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.min_nack_response_delay.nanosec = 0;</code></td>
</tr>
<tr>
<td>18.</td>
<td><code>//consider making non-zero for reliable multicast</code></td>
</tr>
<tr>
<td>19.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.max_nack_response_delay.sec = 0;</code></td>
</tr>
<tr>
<td>20.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.max_nack_response_delay.nanosec = 0;</code></td>
</tr>
<tr>
<td>21.</td>
<td><code>// should be faster than the send rate, but be mindful of OS resolution</code></td>
</tr>
<tr>
<td>22.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;</code></td>
</tr>
<tr>
<td>23.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec = 0;</code></td>
</tr>
<tr>
<td>24.</td>
<td><code>alertReaderWithinThisMs * 1000000;</code></td>
</tr>
<tr>
<td>25.</td>
<td><code>qos-&gt;reliability.max_blocking_time = blockingTime;</code></td>
</tr>
<tr>
<td>26.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.max_heartbeat_retries = 7;</code></td>
</tr>
<tr>
<td>27.</td>
<td><code>qos-&gt;protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;</code></td>
</tr>
</tbody>
</table>

Line 1 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer above): This is the default setting for a writer, shown here strictly for clarity.

Line 2 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer above): Setting the History kind to KEEP_ALL guarantees that no DDS sample is ever lost.

Line 3 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer above): This is the default setting for a writer, shown here strictly for clarity. ‘Push’ mode reliability will yield lower latency than ‘pull’ mode reliability in normal situations where there is no DDS sample loss. (See 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.) Furthermore, it does not matter that each packet sent in response to a command will be small, because our data sent with each command is likely to be small, so that maximizing throughput for this data is not a concern.

Line 5 - Line 10 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer above): For this example, we assume a single writer is writing DDS samples one at a time. If we are not using keys (see 2.4 DDS Samples, Instances, and Keys on page 18), there is no reason to use a queue with room for more than one DDS sample, because we want to resolve a DDS sample completely before moving on to the next. While this negatively impacts throughput, it minimizes memory usage. In this example, a written
11.3.7 Use Cases

DDS sample will remain in the queue until it is acknowledged by all active readers (only 1 for this example).

Line 12 - Line 14 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page): The fastest way for a writer to ensure that a reader is up-to-date is to force an acknowledgment with every DDS sample. We do this by appending a Heartbeat with every DDS sample. This is akin to a certified mail; the writer learns—as soon as the system will allow—whether a reader has received the letter, and can take corrective action if the reader has not. As with certified mail, this model has significant overhead compared to the unreliable case, trading off lower packet efficiency in favor of latency and fast recovery.

Line 16-Line 17 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page): Since the writer takes responsibility for pushing the DDS samples out to the reader, a writer will go into a “heightened alert” mode as soon as the high water mark is reached (which is when any DDS sample is written for this writer) and only come out of this mode when the low water mark is reached (when all DDS samples have been acknowledged for this writer). Note that the selected high and low watermarks are actually the default values.

Line 18-Line 22 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page): When a reader requests a lost DDS sample, we respond to the reader immediately in the interest of faster recovery. If the readers receive packets on unicast, there is no reason to wait, since the writer will eventually have to feed individual readers separately anyway. In case of multicast readers, it makes sense to consider further. If the writer delayed its response enough so that all or most of the readers have had a chance to NACK a DDS sample, the writer may coalesce the requests and send just one packet to all the multicast readers. Suppose that all multicast readers do indeed NACK within approximately 100 μsec. Setting the minimum and maximum delays at 100 μsec will allow the writer to collect all these NACKs and send a single response over multicast. (See 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389 for information on setting min nack response delay and max nack response delay.) Note that Connext DDS relies on the OS to wait for this 100 μsec. Unfortunately, not all operating systems can sleep for such a fine duration. On Windows systems, for example, the minimum achievable sleep time is somewhere between 1 to 20 milliseconds, depending on the version. On VxWorks systems, the minimum resolution of the wait time is based on the tick resolution, which is 1/system clock rate (thus, if the system clock rate is 100 Hz, the tick resolution is 10 milliseconds). On such systems, the achievable minimum wait is actually far larger than the desired wait time. This could have an unintended consequence due to the delay caused by the OS; at a minimum, the time to repair a packet may be longer than you specified.

Line 24-Line 27 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer on the previous page): If a reader drops a DDS sample, the writer recovers by notifying the reader of what it has sent, so that the reader may request resending of the rejected DDS sample. Therefore, the recovery time depends primarily on how quickly the writer pings the reader that has fallen behind. If commands will not be generated faster than one every few seconds, it may be acceptable for the writer to ping the reader several hundred milliseconds after the DDS sample is sent.
• Suppose that the round-trip time of fairly small packets between the writer and the reader application is 50 microseconds, and that the reader does not delay response to a Heartbeat from the writer (see 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570 for how to change this). If a DDS sample is dropped by the network, the writer will ping the reader after a maximum of the OS delay resolution discussed above and alertReaderWithinThisMs (let’s say 10 ms for this example). The reader will request the missing DDS sample immediately, and with the code set as above, the writer will feed the missing DDS sample immediately. Neglecting the processing time on the writer or the reader end, and assuming that this retry succeeds, the time to recover the DDS sample from the original publication time is: alertReaderWithinThisMs + 50 μsec + 25 μsec.

If the OS is capable of micro-sleep, the recovery time can be within 100 μsec, barely noticeable to a human operator. If the OS minimum wait resolution is much larger, the recovery time is dominated by the wait resolution of the OS. Since ergonomic studies suggest that delays in excess of a 0.25 seconds start hampering operations that require low latency data, even a 10 ms limitation seems to be acceptable.

• What if two packets are dropped in a row? Then the recovery time would be

\[2 \times \text{alertReaderWithinThisMs} + 2 \times 50 \, \mu\text{sec} + 25 \, \mu\text{sec}\]  

If alertReaderWithinThisMs is 100 ms, the recovery time now exceeds 200 ms, and can perhaps degrade user experience.

Line 29-Line 30 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer on page 714): What if another command (like another button press) is issued before the recovery? Since we must not drop this new DDS sample, we block the writer until the recovery completes. If alertReaderWithinThisMs is 10 ms, and we assume no more than 7 consecutive drops, the longest time for recovery will be just above (alertReaderWithinThisMs * max_heartbeat_retries), or 70 ms.

So if we set blockingTime to about 80 ms, we will have given enough chance for recovery. Of course, in a dynamic system, a reader may drop out at any time, in which case max_heartbeat_retries will be exceeded, and the unresponsive reader will be dropped by the writer. In either case, the writer can continue writing. Inappropriate values will cause a writer to prematurely drop a temporarily unresponsive (but otherwise healthy) reader, or be stuck trying unsuccessfully to feed a crashed reader. In the unfortunate case where a reader becomes temporarily unresponsive for a duration exceeding (alertReaderWithinThisMs * max_heartbeat_retries), the writer may issue gaps to that reader when it becomes active again; the dropped DDS samples are irrecoverable. So estimating the worst case unresponsive time of all potential readers is critical if DDS sample drop is unacceptable.

Line 33 (Figure 11.10: QoS for an Aperiodic, One-at-a-time Reliable Writer on page 714): Since the command may not be issued for hours or even days on end, there is no reason to keep announcing the writer’s state to the readers.

Figure 11.11: QoS for an Aperiodic, One-at-a-time Reliable Reader on the next page shows how to set the QoS for the reader side, followed by a line-by-line explanation.
Figure 11.11: QoS for an Aperiodic, One-at-a-time Reliable Reader

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. // 1 is ok for normal use. 2 allows fast infinite loop
4. qos->reader_resource_limits.max_samples_per_remote_writer = 2;
5. qos->resource_limits.initial_samples = 2;
6. qos->resource_limits.initial_instances = 1;
7. protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
8. protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec = 0;
9. protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
10. protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;
11. protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;

Line 1-Line 2 (Figure 11.11: QoS for an Aperiodic, One-at-a-time Reliable Reader above): Unlike a writer, the reader’s default reliability setting is best-effort, so reliability must be turned on. Since we don’t want to drop anything, we choose KEEP_ALL history.

Line 4-Line 6 (Figure 11.11: QoS for an Aperiodic, One-at-a-time Reliable Reader above): Since we enforce reliability on each DDS sample, it would be sufficient to keep the queue size at 1, except in the following case: suppose that the reader takes some action in response to the command received, which in turn causes the writer to issue another command right away. Because Connext DDS passes the user data up to the application even before acknowledging the DDS sample to the writer (for minimum latency), the first DDS sample is still pending for acknowledgement in the writer’s queue when the writer attempts to write the second DDS sample, and will cause the writing thread to block until the reader completes processing the first DDS sample and acknowledges it to the writer; all are as they should be. But if you want to run this infinite loop at full throttle, the reader should buffer one more DDS sample. Let’s follow the packets flow under a normal circumstance:

1. The sender application writes DDS sample 1 to the reader. The receiver application processes it and sends a user-level response 1 to the sender application, but has not yet ACK’d DDS sample 1.
2. The sender application writes DDS sample 2 to the receiving application in response to response 1. Because the reader’s queue is 2, it can accept DDS sample 2 even though it may not yet have acknowledged DDS sample 1. Otherwise, the reader may drop DDS sample 2, and would have to recover it later.
3. At the same time, the receiver application acknowledges DDS sample 1, and frees up one slot in the queue, so that it can accept DDS sample 3, which it on its way.

The above steps can be repeated ad-infinitum in a continuous traffic.

Line 7 (Figure 11.11: QoS for an Aperiodic, One-at-a-time Reliable Reader above): Since we are not using keys, there is just one instance.

Line 9-Line 12 (11.3.7 Use Cases on page 713): We choose immediate response in the interest of fastest recovery. In high throughput, multicast scenario, delaying the response (with event thread priority set high
of course) may decrease the likelihood of NACK storm causing a writer to drop some NACKs. This random delay reduces this chance by staggering the NACK response. But the minimum delay achievable once again depends on the OS.

11.3.7.3 Aperiodic, Bursty

Suppose you have aperiodically generated bursts of data, as in the case of a new aircraft approaching an airport. The data may be the same or different, but if they are written by a single writer, the challenge to this writer is to feed all readers as quickly and efficiently as possible when this burst of hundreds or thousands of DDS samples hits the system.

If you use an unreliable writer to push this burst of data, some of them may be dropped over an unreliable transport such as UDP.

If you try to shape the burst according to however much the slowest reader can process, the system throughput may suffer, and places an additional burden of queueing the DDS samples on the sender application.

If you push the data reliably as fast they are generated, this may cost dearly in repair packets, especially to the slowest reader, which is already burdened with application chores.

Connext DDS pull mode reliability offers an alternative in this case by letting each reader pace its own data stream. It works by notifying the reader what it is missing, then waiting for it to request only as much as it can handle. As in the aperiodic one-at-a-time case (11.3.7.2 Aperiodic Use Case: One-at-a-Time on page 713), multicast is supported, but its performance depends on the resolution of the minimum delay supported by the OS. At the cost of greater latency, this model can deliver reliability while using far fewer packets than in the push mode. The writer QoS is given in Figure 11.12: QoS for an Aperiodic, Bursty Writer below, with a line-by-line explanation below.

**Figure 11.12: QoS for an Aperiodic, Bursty Writer**

```plaintext
1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->protocol.push_on_write = DDS_BOOLEAN_FALSE;
4. 
5. //use these hard coded value until you use key
6. qos->resource_limits.initial_instances =
7. qos->resource_limits.max_instances = 1;
8. qos->resource_limits.initial_samples = qos->resource_limits.max_samples
9. = worstBurstInSample;
10. qos->resource_limits.max_samples_per_instance =
11. qos->resource_limits.max_samples;
12. 
13. // piggyback HB not used
14. qos->protocol.rtps_reliable_writer.heartbeats_per_max_samples = 0;
15. 
16. qos->protocol.rtps_reliable_writer.high_watermark = 1;
17. qos->protocol.rtps_reliable_writer.low_watermark = 0;
18. 
19. qos->protocol.rtps_reliable_writer.min_nack_response_delay.sec = 0;
20. qos->protocol.rtps_reliable_writer.min_nack_response_delay.nanosec = 0;
```
21. qos->protocol.rtps_reliable_writer.max_nack_response_delay.sec = 0;
22. qos->protocol.rtps_reliable_writer.max_nack_response_delay.nanosec = 0;
23. qos->reliability.max_blocking_time = blockingTime;
24.
25. // should be faster than the send rate, but be mindful of OS resolution
26. qos->protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;
27. qos->protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec =
28. 1000000;
29. qos->protocol.rtps_reliable_writer.max_heartbeat_retries = 5;
30.
31. // essentially turn off slow HB period
32. qos->protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;

Line 1 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on the previous page): This is the default setting for a writer, shown here strictly for clarity.

Line 2 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on the previous page): Since we do not want any data lost, we want the History kind set to KEEP_ALL.

Line 3 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on the previous page): The default Connext DDS reliable writer will push, but we want the reader to pull instead.

Line 5-Line 11 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on the previous page): We assume a single instance, in which case the maximum DDS sample count will be the same as the maximum DDS sample count per writer. In contrast to the one-at-a-time case discussed in 11.3.7.2 Aperiodic Use Case: One-at-a-Time on page 713, the writer’s queue is large; as big as the burst size in fact, but no more because this model tries to resolve a burst within a reasonable period, to be computed shortly. Of course, we could block the writing thread in the middle of the burst, but that might complicate the design of the sending application.

Line 13-Line 14 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on the previous page): By a ‘piggy-back’ Heartbeat, we mean only a Heartbeat that is appended to data being pushed from the writer. Strictly speaking, the writer will also append a Heartbeat with each reply to a reader’s lost DDS sample request, but we call that a ‘framing’ Heartbeat. Since data is pulled, heartbeats_per_max_samples is ignored.

Line 16-Line 17 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on the previous page): Similar to the previous aperiodic writer, this writer spends most of its time idle. But as the name suggests, even a single new DDS sample implies more DDS sample to follow in a burst. Putting the writer into a fast mode quickly will allow readers to be notified soon. Only when all DDS samples have been delivered, the writer can rest.

Line 19- Line 23 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on the previous page): Similar to the one-at-a-time case, there is no reason to delay response with only one reader. In this case, we can estimate the time to resolve a burst with only a few parameters. Let’s say that the reader figures it can safely receive and process 20 DDS samples at a time without being overwhelmed, and that the time it takes a writer to fetch these 20 DDS samples and send a single packet containing these 20 DDS samples, plus the time it takes a reader to receive and process these DDS samples, and send another request back to the writer for the next 20 DDS samples is 11 ms. Even on the same hardware, if the reader’s processing time can be
reduced, this time will decrease; other factors such as the traversal time through Connext DDS and the transport are typically in microseconds range (depending on machines of course).

For example, let’s also say that the worst case burst is 1000 DDS samples. The writing thread will of course not block because it is merely copying each of the 1000 DDS samples to the Connext DDS queue on the writer side; on a typical modern machine, the act of writing these 1000 DDS samples will probably take no more than a few ms. But it would take at least 1000/20 = 50 resend packets for the reader to catch up to the writer, or 50 times 11 ms = 550 ms. Since the burst model deals with one burst at a time, we would expect that another burst would not come within this time, and that we are allowed to block for at least this period. Including a safety margin, it would appear that we can comfortably handle a burst of 1000 every second or so.

But what if there are multiple readers? The writer would then take more time to feed multiple readers, but with a fast transport, a few more readers may only increase the 11 ms to only 12 ms or so. Eventually, however, the number of readers will justify the use of multicast. Even in pull mode, Connext DDS supports multicast by measuring how many multicast readers have requested DDS sample repair. If the writer does not delay response to NACK, then repairs will be sent in unicast. But a suitable NACK delay allows the writer to collect potentially NACKs from multiple readers, and feed a single multicast packet. But as discussed in 11.3.7.2 Aperiodic Use Case: One-at-a-Time on page 713, by delaying reply to coalesce response, we may end up waiting much longer than desired. On a Windows system with 10 ms minimum sleep achievable, the delay would add at least 10 ms to the 11 ms delay, so that the time to push 1000 DDS samples now increases to 50 times 21 ms = 1.05 seconds. It would appear that we will not be able to keep up with incoming burst if it came at roughly 1 second, although we put fewer packets on the wire by taking advantage of multicast.

Line 25-Line 28 (11.3.7 Use Cases on page 713): We now understand how the writer feeds the reader in response to the NACKs. But how does the reader realize that it is behind? The writer notifies the reader with a Heartbeat to kick-start the exchange. Therefore, the latency will be lower bound by the writer’s fast heartbeat period. If the application is not particularly sensitive to latency, the minimum wait time supported by the OS (10 ms on Windows systems, for example) might be a reasonable value.

Line 29 (Figure 11.12: QoS for an Aperiodic, Bursty Writer on page 718): With a fast heartbeat period of 50 ms, a writer will take 500 ms (50 ms times the default max_heartbeat_retries of 10) to write-off an unresponsive reader. If a reader crashes while we are writing a lot of DDS samples per second, the writer queue may completely fill up before the writer has a chance to drop the crashed reader. Lowering max_heartbeat_retries will prevent that scenario.

Line 31-Line 32 (Figure 11.12: QoS for an Aperiodic, Bursty Reader on page 718): For an aperiodic writer, turning off slow periodic Heartbeats will remove unwanted traffic from the network.

Figure 11.13: QoS for an Aperiodic, Bursty Reader on the next page shows example code for a corresponding aperiodic, bursty reader.
11.3.7 Use Cases

Figure 11.13: QoS for an Aperiodic, Bursty Reader

```
1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->resource_limits.initial_samples =
4. qos->resource_limits.max_samples =
5. qos->reader_resource_limits.max_samples_per_remote_writer = 32;
6. // use these hard coded value until you use key
7. qos->resource_limits.max_samples_per_instance =
8. qos->resource_limits.max_samples =
9. qos->resource_limits.max_instances =
10. qos->resource_limits.max_instances = 1;
11. // the writer probably has more for the reader; ask right away
12. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
13. qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;
14. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
15. qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec = 0;
```

Line 1-Line 2 (Figure 11.13: QoS for an Aperiodic, Bursty Reader above): Unlike a writer, the reader’s default reliability setting is best-effort, so reliability must be turned on. Since we don’t want to drop anything, we choose KEEP_ALL for the History QoS kind.

Line 3-Line 5 (Figure 11.13: QoS for an Aperiodic, Bursty Reader above): Unlike the writer, the reader’s queue can be kept small, since the reader is free to send ACKs for as much as it wants anyway. In general, the larger the queue, the larger the packet needs to be, and the higher the throughput will be. When the reader NACKs for lost DDS sample, it will only ask for this much.

Line 7-Line 11 (Figure 11.13: QoS for an Aperiodic, Bursty Reader above): We do not use keys in this example.

Line 13-Line 17 (Figure 11.13: QoS for an Aperiodic, Bursty Reader above): We respond immediately to catch up as soon as possible. When there are many readers, this may cause a NACK storm, as discussed in the reader code for one-at-a-time reliable reader.

11.3.7.4 Periodic

In a periodic reliable model, we can use the writer and the reader queue to keep the data flowing at a smooth rate. The data flows from the sending application to the writer queue, then to the transport, then to the reader queue, and finally to the receiving application. Unless the sending application or any one of the receiving applications becomes unresponsive (including a crash) for a noticeable duration, this flow should continue uninterrupted.

The latency will be low in most cases, but will be several times higher for the recovered and many subsequent DDS samples. In the event of a disruption (e.g., loss in transport, or one of the readers becoming temporarily unresponsive), the writer’s queue level will rise, and may even block in the worst case. If the writing thread must not block, the writer’s queue must be sized sufficiently large to deal with any fluctuation in the system. Figure 11.14: QoS for a Periodic Reliable Writer on the next page shows an example, with line-by-line analysis below.
Figure 11.14: QoS for a Periodic Reliable Writer

1. qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
2. qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
3. qos->protocol.push_on_write = DDS_BOOLEAN_TRUE;
4. //use these hard coded value until you use key
5. qos->resource_limits.initial_instances =
6. qos->resource_limits.max_instances = 1;
7. int unresolvedSamplePerRemoteWriterMax =
8. worstCaseApplicationDelayTimeInMs * dataRateInHz / 1000;
9. qos->resource_limits.max_samples = unresolvedSamplePerRemoteWriterMax;
10. qos->resource_limits.initial_samples = qos->resource_limits.max_samples/2;
11. qos->resource_limits.max_samples_per_instance =
12. qos->resource_limits.max_samples;
13. int piggybackEvery = 8;
14. qos->protocol.rtps_reliable_writer.heartbeats_per_max_samples =
15. qos->resource_limits.max_samples / piggybackEvery;
16. qos->protocol.rtps_reliable_writer.high_watermark = piggybackEvery * 4;
17. qos->protocol.rtps_reliable_writer.low_watermark = piggybackEvery * 2;
18. qos->reliability.max_blocking_time = blockingTime;
19. qos->protocol.rtps_reliable_writer.min_nack_response_delay.sec = 0;
20. qos->protocol.rtps_reliable_writer.min_nack_response_delay.nanosec = 0;
21. qos->protocol.rtps_reliable_writer.max_nack_response_delay.sec = 0;
22. qos->protocol.rtps_reliable_writer.max_nack_response_delay.nanosec = 0;
23. qos->protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;
24. qos->protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec =
25. alertReaderWithinThisMs * 1000000;
26. qos->protocol.rtps_reliable_writer.max_heartbeat_retries = 7;
27. // essentially turn off slow HB period
28. qos->protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;

Line 1 (Figure 11.14: QoS for a Periodic Reliable Writer above): This is the default setting for a writer, shown here strictly for clarity.

Line 2 (Figure 11.14: QoS for a Periodic Reliable Writer above): Since we do not want any data lost, we set the History kind to KEEP_ALL.

Line 3 (Figure 11.14: QoS for a Periodic Reliable Writer above): This is the default setting for a writer, shown here strictly for clarity. Pushing will yield lower latency than pulling.

Line 5-Line 7 (Figure 11.14: QoS for a Periodic Reliable Writer above): We do not use keys in this example, so there is only one instance.

Line 9-Line 11 (Figure 11.14: QoS for a Periodic Reliable Writer above): Though a simplistic model of queue, this is consistent with the idea that the queue size should be proportional to the data rate and the worst case jitter in communication.
Line 12 (Figure 11.14: QoS for a Periodic Reliable Writer on the previous page): Even though we have sized the queue according to the worst case, there is a possibility for saving some memory in the normal case. Here, we initially size the queue to be only half of the worst case, hoping that the worst case will not occur. When it does, Connext DDS will keep increasing the queue size as necessary to accommodate new DDS samples, until the maximum is reached. So when our optimistic initial queue size is breached, we will incur the penalty of dynamic memory allocation. Furthermore, you will wind up using more memory, as the initially allocated memory will be orphaned (note: does not mean a memory leak or dangling pointer); if the initial queue size is $M_i$ and the maximal queue size is $M_m$, where $M_m = M_i * 2^n$, the memory wasted in the worst case will be $(M_m - 1) \times \text{sizeof}(\text{DDS sample})$ bytes. Note that the memory allocation can be avoided by setting the initial queue size equal to its max value.

Line 13-Line 14 (Figure 11.14: QoS for a Periodic Reliable Writer on the previous page): If there is only one instance, maximum DDS samples per instance is the same as maximum DDS samples allowed.

Line 16-Line 18 (Figure 11.14: QoS for a Periodic Reliable Writer on the previous page): Since we are pushing out the data at a potentially rapid rate, the piggyback heartbeat will be useful in letting the reader know about any missing DDS samples. The piggybackEvery can be increased if the writer is writing at a fast rate, with the cost that more DDS samples will need to queue up for possible resend. That is, you can consider the piggyback heartbeat to be taking over one of the roles of the periodic heartbeat in the case of a push. So sending fewer DDS samples between piggyback heartbeats is akin to decreasing the fast heartbeat period seen in previous sections. Please note that we cannot express piggybackEvery directly as its own QoS, but indirectly through the maximum DDS samples.

Line 20-Line 22 (Figure 11.14: QoS for a Periodic Reliable Writer on the previous page): If piggybackEvery was exactly identical to the fast heartbeat, there would be no need for fast heartbeat or the high watermark. But one of the important roles for the fast heartbeat period is to allow a writer to abandon inactive readers before the queue fills. If the high watermark is set equal to the queue size, the writer would not doubt the status of an unresponsive reader until the queue completely fills—blocking on the next write (up to blockingTime). By lowering the high watermark, you can control how vigilant a writer is about checking the status of unresponsive readers. By scaling the high watermark to piggybackEvery, the writer is expressing confidence that an alive reader will respond promptly within the time it would take a writer to send 4 times piggybackEvery DDS samples. If the reader does not delay the response too long, this would be a good assumption. Even if the writer estimated on the low side and does go into fast mode (suspecting that the reader has crashed) when a reader is temporarily unresponsive (e.g., when it is performing heavy computation for a few milliseconds), a response from the reader in question will resolve any doubt, and data delivery can continue uninterrupted. As the reader catches up to the writer and the queue level falls below the low watermark, the writer will pop out to the normal, relaxed mode.

Line 24-Line 28 (Figure 11.14: QoS for a Periodic Reliable Writer on the previous page): When a reader is behind (including a reader whose Durability QoS is non-VOLATILE and therefore needs to catch up to the writer as soon as it is created), how quickly the writer responds to the reader’s request will determine the catch-up rate. While a multicast writer (that is, a writer with multicast readers) may consider delaying for some time to take advantage of coalesced multicast packets. Keep in mind the OS delay resolution issue discussed in the previous section.
Line 30-Line 33 (Figure 11.14: QoS for a Periodic Reliable Writer on page 722): The fast heartbeat mechanism allows a writer to detect a crashed reader and move along with the remaining readers when a reader does not respond to any of the `max_heartbeat_retries` number of heartbeats sent at the `fast_heartbeat_period` rate. So if you want a more cautious writer, decrease either numbers; conversely, increasing either number will result in a writer that is more reluctant to write-off an unresponsive reader.

Line 35-Line 36 (Figure 11.14: QoS for a Periodic Reliable Writer on page 722): Since this a periodic model, a separate periodic heartbeat to notify the writer’s status would seem unwarranted; the piggyback heartbeat sent with DDS samples takes over that role.

Figure 11.15: QoS for a Periodic Reliable Reader below shows how to set the QoS for a matching reader, followed by a line-by-line explanation.

**Figure 11.15: QoS for a Periodic Reliable Reader**

1. `qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;`
2. `qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;`
3. `qos->resource_limits.initial_samples =`  
4. `qos->resource_limits.max_samples =`  
5. `qos->reader_resource_limits.max_samples_per_remote_writer =`  
6. `((2*piggybackEvery - 1) + dataRateInHz * delayInMs / 1000);`  
7. `//use these hard coded value until you use key`  
8. `qos->resource_limits.max_samples_per_instance =`  
9. `qos->resource_limits.max_samples;`  
10. `qos->resource_limits.initial_instances =`  
11. `qos->resource_limits.max_instances = 1;`  
12. `qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;`  
13. `qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;`  
14. `qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;`  
15. `qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec = 0;`  

Line 1-Line 2 (Figure 11.15: QoS for a Periodic Reliable Reader above): Unlike a writer, the reader’s default reliability setting is best-effort, so reliability must be turned on. Since we don’t want to drop anything, we choose `KEEP_ALL` for the History QoS.

Line 3-Line 6 (Figure 11.15: QoS for a Periodic Reliable Reader above) Unlike the writer, the reader queue is sized not according to the jitter of the reader, but rather how many DDS samples you want to cache speculatively in case of a gap in sequence of DDS samples that the reader must recover. Remember that a reader will stop giving a sequence of DDS samples as soon as an unintended gap appears, because the definition of strict reliability includes in-order delivery. If the queue size were 1, the reader would have no choice but to drop all subsequent DDS samples received until the one being sought is recovered. Connext DDS uses speculative caching, which minimizes the disruption caused by a few dropped DDS samples. Even for the same duration of disruption, the demand on reader queue size is greater if the writer will send more rapidly. In sizing the reader queue, we consider two factors that comprise the DDS sample recovery time:
• How long it takes a reader to request a resend to the writer.

The piggyback heartbeat tells a reader about the writer’s state. If only DDS samples between two piggybacked DDS samples are dropped, the reader must cache piggybackEvery DDS samples before asking the writer for resend. But if a piggybacked DDS sample is also lost, the reader will not get around to asking the writer until the next piggybacked DDS sample is received. Note that in this worst case calculation, we are ignoring stand-alone heartbeats (i.e., not piggybacked heartbeat from the writer). Of course, the reader may drop any number of heartbeats, including the stand-alone heartbeat; in this sense, there is no such thing as the absolute worst case—just reasonable worst case, where the probability of consecutive drops is acceptably low. For the majority of applications, even two consecutive drops is unlikely, in which case we need to cache at most \((2 \times \text{ piggybackEvery } - 1)\) DDS samples before the reader will ask the writer to resend, assuming no delay (Line 14-Line 17, Figure 11.15: QoS for a Periodic Reliable Reader on the previous page).

• How long it takes for the writer to respond to the request.

Even ignoring the flight time of the resend request through the transport, the writer takes a finite time to respond to the repair request—mostly if the writer delays reply for multicast readers. In case of immediate response, the processing time on the writer end, as well as the flight time of the messages to and from the writer do not matter unless very larger data rate; that is, it is the product term that matters. In case the delay for multicast is random (that is, the minimum and the maximum delay are not equal), one would have to use the maximum delay to be conservative.

Line 8-Line 12 (Figure 11.15: QoS for a Periodic Reliable Reader on the previous page): Since we are not using keys, there is just one instance.

Line 14-Line 17 (Figure 11.15: QoS for a Periodic Reliable Reader on the previous page): If we are not using multicast, or the number of readers being fed by the writer, there is no reason to delay.

11.4 Auto Throttling for DataWriter Performance—Experimental Feature

Auto Throttling is an experimental feature that allows you to configure a DataWriter to automatically adjust its writing rate and send window size to provide the best latency/throughput tradeoff as system conditions change.

When DataWriters and DataReaders are configured to be reliable, DDS samples that did not reach the matched DataReaders for any reason (such as network drops or sample rejection by the DataReader) are repaired automatically by Connext DDS. However, the repair path consumes bandwidth and increases latency. A high number of repaired DDS samples can reduce the throughput and increase the communication latency. With Auto Throttling, the number of repair DDS samples is reduced by using feedback provided by DataReaders in terms of ACK and NACK messages to adjust the DataWriter’s write rate and send window size.

To configure Auto Throttling, use the following properties:
dds.domain_participant.auto_throttle.enable: Configures the DomainParticipant to gather internal measurements (during DomainParticipant creation) that are required for the Auto Throttle feature. This allows DataWriters belonging to this DomainParticipant to use the Auto Throttle feature. Default: false.

dds.data_writer.auto_throttle.enable: Enables automatic throttling in the DataWriter so it can automatically adjust the writing rate and the send window size; this minimizes the need for repair DDS samples and improves latency. Default: false.

Note: This property takes effect only in DataWriters that belong to a DomainParticipant that has set the property dds.domain_participant.auto_throttle.enable (described above) to true.

When Auto throttling is enabled, the size of the send window size is adjusted within the interval [min_send_window_size, max_send_window_size] configured in 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.
Chapter 12 Collaborative DataWriters

The Collaborative DataWriters feature allows you to have multiple DataWriters publishing DDS samples from a common logical data source. The DataReaders will combine the DDS samples coming from these DataWriters in order to reconstruct the correct order in which they were produced at the source. This combination process for the DataReaders can be configured using the 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370. It requires the middleware to provide a way to uniquely identify every DDS sample published in a DDS domain independently of the actual DataWriter that published the DDS sample.

In Connext DDS, every modification (DDS sample) to the global dataspace made by a DataWriter within a DDS domain is identified by a pair (virtual GUID, sequence number).

The virtual GUID (Global Unique Identifier) is a 16-byte character identifier associated with the logical data source. DataWriters can be assigned a virtual GUID using virtual_guid in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

The virtual sequence number is a 64-bit integer that identifies changes within the logical data source.

Several DataWriters can be configured with the same virtual GUID. If each of these DataWriters publishes a DDS sample with sequence number '0', the DDS sample will only be received once by the DataReaders subscribing to the content published by the DataWriters (see Figure 12.1: Global Dataspace Changes on the next page).
12.1 Collaborative DataWriters Use Cases

- Ordered delivery of DDS samples in high availability scenarios

One example of this is RTI Persistence Service\(^1\). When a late-joining DataReader configured with 7.5.9 DURABILITY QosPolicy on page 411 set to PERSISTENT or TRANSIENT joins a DDS domain, it will start receiving DDS samples from multiple DataWriters. For example, if the original DataWriter is still alive, the newly created DataReader will receive DDS samples from the original DataWriter and one or more RTI Persistence Service DataWriters (PRSTDataWriters).

- Ordered delivery of DDS samples in load-balanced scenarios

Multiple instances of the same application can work together to process and deliver DDS samples. When the DDS samples arrive through different data-paths out of order, the DataReader will be able to reconstruct the order at the source. An example of this is when multiple instances of RTI Persistence Service are used to persist the data. Persisting data to a database on disk can impact performance. By dividing the workload (e.g., DDS samples larger than 10 are persisted by Persistence Service 1, DDS samples smaller or equal to 10 are persisted by Persistence Service 2) across different instances of RTI Persistence Service using different databases the user can improve scalability and performance.

- Ordered delivery of DDS samples with Group Ordered Access

The Collaborative DataWriters feature can also be used to configure the DDS sample ordering process when the Subscriber is configured with 7.4.6 PRESENTATION QosPolicy on page 362.

---

\(^1\)For more information on Persistence Service, see Part 8: RTI Persistence Service on page 1179.
access_scope set to GROUP. In this case, the Subscriber must deliver in order the DDS samples published by a group of DataWriters that belong to the same Publisher and have access_scope set to GROUP.

Figure 12.2: Load-Balancing with Persistence Service

12.2 DDS Sample Combination (Synchronization) Process in a DataReader

A DataReader will deliver a DDS sample (VGUIDn, VSNm) to the application only when if one of the following conditions is satisfied:

- (VGUIDn, VSNm-1) has already been delivered to the application.
- All the known DataWriters publishing VGUIDn have announced that they do not have (VGUIDn, VSNm-1).
- None of the known DataWriters publishing VGUIDn have announced potential availability of (VGUIDn, VSNm-1) and a configurable timeout (max_data_availability_waiting_time) expires.

For additional details on how the reconstruction process works see the 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370.
12.3 Configuring Collaborative DataWriters

12.3.1 Associating Virtual GUIDs with DDS Data Samples

There are two ways to associate a virtual GUID with the DDS samples published by a DataWriter:

- Per DataWriter: Using virtual_guid in 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.

- Per DDS Sample: By setting the writer_guid in the identity field of the WriteParams_t structure provided to the write_w_params operation (see 7.3.8 Writing Data on page 309). Since the writer_guid can be set per DDS sample, the same DataWriter can potentially write DDS samples from independent logical data sources. One example of this is RTI Persistence Service where a single persistence service DataWriter can write DDS samples on behalf of multiple original DataWriters.

12.3.2 Associating Virtual Sequence Numbers with DDS Data Samples

You can associate a virtual sequence number with a DDS sample published by a DataWriter by setting the sequence_number in the identity field of the WriteParams_t structure provided to the write_w_params operation (see 7.3.8 Writing Data on page 309). Virtual sequence numbers for a given virtual GUID must be strictly monotonically increasing. If you try to write a DDS sample with a sequence number less than or equal to the last sequence number, the write operation will fail.

12.3.3 Specifying which DataWriters will Deliver DDS Samples to the DataReader from a Logical Data Source

The required_matched_endpoint_groups field in the 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370 can be used to specify the set of DataWriter groups that are expected to provide DDS samples for the same data source (virtual GUID). The quorum count in a group represents the number of DataWriters that must be discovered for that group before the DataReader is allowed to provide non-consecutive DDS samples to the application.

A DataWriter becomes a member of an endpoint group by configuring the role_name in 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.

12.3.4 Specifying How Long to Wait for a Missing DDS Sample

A DataReader’s 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370 specifies how long to wait for a missing DDS sample. For example, this is important when the first DDS sample is received: how long do you wait to determine the lowest sequence number available in the system?

- The max_data_availability_waiting_time defines how much time to wait before delivering a DDS sample to the application without having received some of the previous DDS samples.
The `max_endpoint_availability_waiting_time` defines how much time to wait to discover `DataWriters` providing DDS samples for the same data source (virtual GUID).

### 12.4 Collaborative DataWriters and Persistence Service

The `DataWriters` created by persistence service are automatically configured to do collaboration:

- Every DDS sample published by the `Persistence Service DataWriter` keeps its original identity.
- `Persistence Service` associates the role name `PERSISTENCE_SERVICE` with all the `DataWriters` that it creates. You can overwrite that setting by changing the `DataWriter QoS` configuration in persistence service.

For more information, see Part 8: RTI Persistence Service on page 1179.
Chapter 13 Mechanisms for Achieving Information Durability and Persistence

13.1 Introduction

Durable Writer History and Durable Reader State are deprecated as of release 6.1.1 and will be removed in a future release. Use Part 8: RTI Persistence Service on page 1179 to persist your data instead.

Connext DDS offers the following mechanisms for achieving durability and persistence:

- **Durable Writer History** This feature allows a *DataWriter* to persist its historical cache, perhaps locally, so that it can survive shutdowns, crashes and restarts. When an application restarts, each *DataWriter* that has been configured to have durable writer history automatically loads all of the data in this cache from disk and can carry on sending data as if it had never stopped executing. To the rest of the system, it will appear as if the *DataWriter* had been temporarily disconnected from the network and then reappeared.

- **Durable Reader State** This feature allows a *DataReader* to persist its state and remember which data it has already received. When an application restarts, each *DataReader* that has been configured to have durable reader state automatically loads its state from disk and can carry on receiving data as if it had never stopped executing. Data that had already been received by the *DataReader* before the restart will be suppressed so that it is not even sent over the network.

- **Data Durability** This feature is a full implementation of the OMG DDS Persistence Profile. The 7.5.9 DURABILITY QosPolicy on page 411 allows an application to configure a *DataWriter* so that the information written by the *DataWriter* survives beyond the lifetime of the *DataWriter*. In this manner, a late-joining *DataReader* can subscribe to and receive the information even after the *DataWriter* application is no longer executing. To use this feature,
13.1.1 Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History)

you need Persistence Service, a separate application described in Introduction to RTI Persistence Service (Chapter 45 on page 1180).

These features can be configured separately or in combination. To use Durable Writer State and Durable Reader State, you need a relational database, which is not included with Connext DDS. Persistence Service does not require a database when used in TRANSIENT mode (see 13.5.1 RTI Persistence Service on page 749) or in PERSISTENT mode with file-system storage (see 13.5.1 RTI Persistence Service on page 749 and 46.5 Configuring Remote Administration on page 1188). See RTI Connext DDS Core Libraries Database Setup for a list of supported relational databases and setup instructions.

To understand how these features interact we will examine the behavior of the system using the following scenarios:

- 13.1.1 Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History) below
- 13.1.2 Scenario 2: DataReader Restarts While DataWriter Stays Up (Durable Reader State) on page 735
- 13.1.3 Scenario 3. DataReader Joins after DataWriter Leaves Domain (Durable Data) on page 736

13.1.1 Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History)

In this scenario, a DomainParticipant joins the domain, creates a DataWriter and writes some data, then the DataWriter shuts down (gracefully or due to a fault). The DataWriter restarts and a DataReader joins the domain. Depending on whether the DataWriter is configured with durable history, the late-joining DataReader may or may not receive the data published already by the DataWriter before it restarted. This is illustrated in Figure 13.1: Durable Writer History on the next page. For more information, see 13.3 Durable Writer History on page 739.
13.1.1 Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History)

Figure 13.1: Durable Writer History

Without Durable Writer History: the late-joining DataReader will not receive data (a and b) that was published before the DataWriter’s restart.

With Durable Writer History: the restarted DataWriter will recover its history and deliver its data to the late-joining DataReader.
13.1.2 Scenario 2: DataReader Restarts While DataWriter Stays Up (Durable Reader State)

In this scenario, two DomainParticipants join a domain; one creates a DataWriter and the other a DataReader on the same Topic. The DataWriter publishes some data ("a" and "b") that is received by the DataReader. After this, the DataReader shuts down (gracefully or due to a fault) and then restarts—all while the DataWriter remains present in the domain.

Depending on whether the DataReader is configured with Durable Reader State, the DataReader may or may not receive a duplicate copy of the data it received before it restarted. This is illustrated in Figure 13.2: Durable Reader State below. For more information, see 13.4 Durable Reader State on page 744.

Figure 13.2: Durable Reader State

Without Durable Reader State: the DataReader will receive the data that was already received before the restart.

With Durable Reader State: the DataReader remembers that it already received the data and does not request it again.
13.1.3 Scenario 3. DataReader Joins after DataWriter Leaves Domain (Durable Data)

In this scenario, a DomainParticipant joins a domain, creates a DataWriter, publishes some data on a Topic and then shuts down (gracefully or due to a fault). Later, a DataReader joins the domain and subscribes to the data. Persistence Service is running.

Depending on whether Durable Data is enabled for the Topic, the DataReader may or may not receive the data previous published by the DataWriter. This is illustrated in Figure 13.3: Durable Data on the next page. For more information, see 13.5 Data Durability on page 749.
Figure 13.3: Durable Data

Without Durable Data:
the late-joining DataReader
will not receive data (a and b)
that was published before the
DataWriter quit.

With Durable Data:
Persistence Service
remembers what data was
published and delivers it to
the late-joining DataReader.

This third scenario is similar to 13.1.1 Scenario 1. DataReader Joins after DataWriter Restarts (Durable Writer History) on page 733 except that in this case the DataWriter does not need to restart for the DataReader to get the data previously written by the DataWriter. This is because Persistence Service acts as an intermediary that stores the data so it can be given to late-joining DataReaders.

13.2 Durability and Persistence Based on Virtual GUIDs

Every modification to the global dataspace made by a DataWriter is identified by a pair (virtual GUID, sequence number).
13.2 Durability and Persistence Based on Virtual GUIDs

- The virtual GUID (Global Unique Identifier) is a 16-byte character identifier associated with a DataWriter or DataReader; it is used to uniquely identify this entity in the global data space.
- The sequence number is a 64-bit identifier that identifies changes published by a specific DataWriter.

Several DataWriters can be configured with the same virtual GUID. If each of these DataWriters publishes a sample with sequence number '0', the sample will only be received once by the DataReaders subscribing to the content published by the DataWriters (see Figure 13.4: Global Dataspace Changes below).

**Figure 13.4: Global Dataspace Changes**

![Diagram of global dataspace changes](image)

Additionally, Connext DDS uses the virtual GUID to associate a persisted state (state in permanent storage) to the corresponding Entity.

For example, the history of a DataWriter will be persisted in a database table with a name generated from the virtual GUID of the DataWriter. If the DataWriter is restarted, it must have associated the same virtual GUID to restore its previous history.

Likewise, the state of a DataReader will be persisted in a database table whose name is generated from the DataReader virtual GUID (see Figure 13.5: History/State Persistence Based on Virtual GUID on the next page).
13.3 Durable Writer History

Figure 13.5: History/State Persistence Based on Virtual GUID

- A DataWriter’s virtual GUID can be configured using the member virtual_guid in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389.
- A DataReader’s virtual GUID can be configured using the member virtual_guid in the 8.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) on page 570.

The DDS_PublicationBuiltinTopicData and DDS_SubscriptionBuiltinTopicData structures include the virtual GUID associated with the discovered publication or subscription (see 18.2 Built-in DataReaders on page 832).

13.3 Durable Writer History

The 7.5.9 DURABILITY QosPolicy on page 411 controls whether or not, and how, published samples are stored by the DataWriter application for DataReaders that are found after the samples were initially written. The samples stored by the DataWriter constitute the DataWriter’s history.

Connext DDS provides the capability to make the DataWriter history durable, by persisting its content in a relational database. This makes it possible for the history to be restored when the DataWriter restarts. See RTI Connext DDS Core Libraries Database Setup for a list of supported relational databases and setup instructions.

The association between the history stored in the database and the DataWriter is done using the virtual GUID.
13.3.1 Durable Writer History Use Case

The following use case describes the durable writer history functionality:

1. A DataReader receives two samples with sequence number 1 and 2 published by a DataWriter with virtual GUID 1.

2. The process running the DataWriter is stopped and a new late-joining DataReader is created.

   The new DataReader with virtual GUID 2 does not receive samples 1 and 2 because the original DataWriter has been destroyed. If the samples must be available to late-joining DataReaders after the DataWriter deletion, you can use Persistence Service, described in Introduction to RTI Persistence Service (Chapter 45 on page 1180).

3. The DataWriter is restarted using the same virtual GUID.

   After being restarted, the DataWriter restores its history. The late-joining DataReader will receive samples 1 and 2 because they were not received previously. The DataReader with virtual GUID 1 will not receive samples 1 and 2 because it already received them.
4. The *DataWriter* publishes two new samples.

The two new samples with sequence numbers 3 and 4 will be received by both *DataReaders*.

### 13.3.2 How To Configure Durable Writer History

*Connext DDS* allows a *DataWriter*’s history to be stored in a relational database that provides an ODBC driver.

For each *DataWriter* history that is configured to be durable, *Connext DDS* will create a maximum of two tables:

- The first table is used to store the samples associated with the writer history. The name of that table is WS<32 uuencoding of the writer virtual GUID>.
- The second table is only created for keyed-topic and it is used to store the instances associated with the writer history. The name of the second table is WI<32 uuencoding of the writer virtual GUID>.

To configure durable writer history, use the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 associated with *DataWriters* and *DomainParticipants*.

A ‘durable writer history’ property defined in the *DomainParticipant* will be applicable to all the *DataWriters* belonging to the *DomainParticipant* unless it is overwritten by the *DataWriter*. Table 13.1 Durable Writer History Properties lists the supported ‘durable writer history’ properties.

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.data_writer.history.plugin_name</td>
<td>Required. Must be set to <em>dds.data_writer.history.odbc_pluginbuiltin</em> to enable durable writer history in the <em>DataWriter</em>.</td>
</tr>
</tbody>
</table>
### Table 13.1 Durable Writer History Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.data_writer.history.odbc_plugin.dsn</td>
<td>Required. The ODBC DSN (Data Source Name) associated with the database where the writer history must be persisted.</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.driver</td>
<td>Tells Connext DDS which ODBC driver to load. If the property is not specified, Connext DDS will try to use the standard ODBC driver manager library (UnixOdbc on Linux systems, the Windows ODBC driver manager on Windows systems).</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.username</td>
<td>Configures the username/password used to connect to the database. Default: No password or username</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.password</td>
<td></td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.shared</td>
<td>When set to 1, Connext DDS will create a single connection per DSN that will be shared across DataWriters within the same Publisher. A DataWriter can be configured to create its own database connection by setting this property to 0 (the default).</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.instance_cache_max_size</td>
<td>These properties configure the resource limits associated with the ODBC writer history caches. To minimize the number of accesses to the database, Connext DDS uses two caches, one for samples and one for instances. The initial size and the maximum size of these caches are configured using these properties.</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.instance_cache_init_size</td>
<td>The resource limits, initial_instances, max_instances, initial_samples, max_samples, and max_samples_per_instance defined in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451 are used to configure the maximum number of samples and instances that can be stored in the relational database. Defaults:</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.sample_cache_max_size</td>
<td>instance_cache_max_size: max_instances in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.sample_cache_init_size</td>
<td>instance_cache_init_size: initial_instances in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451</td>
</tr>
<tr>
<td>dds.data_writer.history.odbc_plugin.restore</td>
<td>This property indicates whether or not the persisted writer history must be restored once the DataWriter is restarted. If this property is 0, the content of the database associated with the DataWriter being restarted will be deleted. If it is 1, the DataWriter will restore its previous state from the database content. Default: 1</td>
</tr>
</tbody>
</table>

**Note:** If **in_memory_state** (see below in this table) is 1, **instance_cache_max_size** is always equal to **max_instances** in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451—it cannot be changed.
13.3.2 How To Configure Durable Writer History

Table 13.1 Durable Writer History Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.data_writer_.history.odbc_plugin.in_memory_state</td>
<td>This property determines how much state will be kept in memory by the ODBC writer history in order to avoid accessing the database. If this property is 1, then the property <code>instance_cache_max_size</code> (see above in this table) is always equal to <code>max_instances</code> in 7.5.22 RESOURCE_LIMITS QosPolicy on page 451—it cannot be changed. In addition, the ODBC writer history will keep in memory a fixed state overhead of 24 bytes per sample. This mode provides the best ODBC writer history performance. However, the restore operation will be slower and the maximum number of samples that the writer history can manage is limited by the available physical memory. If it is 0, all the state will be kept in the underlying database. In this mode, the maximum number of samples in the writer history is not limited by the physical memory available. Default: 1</td>
</tr>
</tbody>
</table>

Durable Writer History is not supported for Multi-channel DataWriters (see Multi-channel DataWriters (Chapter 20 on page 896)) or when Batching is enabled (see 7.5.2 BATCH QosPolicy (DDS Extension) on page 374); an error is reported if this type of DataWriter tries to configure Durable Writer History.

See also: 13.4 Durable Reader State on the next page.

Example C++ Code

```cpp
/* Get default QoS */
...
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
    "dds.data_writer.history.plugin_name",
    "dds.data_writer.history.odbc_plugin_builtin",
    DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
    "dds.data_writer.history.odbc_plugin.dsn",
    "<user DSN>",
    DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
    "dds.data_writer.history.odbc_plugin.driver",
    "<ODBC library>",
    DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property (writerQos.property,
    "dds.data_writer.history.odbc_plugin.shared",
    "<0|1>",
    DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}```
13.4 Durable Reader State

**Durable reader state** allows a *DataReader* to locally store its state in disk and remember the data that has already been processed by the application. When an application restarts, each *DataReader* configured to have durable reader state automatically reads its state from disk. Data that has already been processed by the application before the restart will not be provided to the application again.

**Important:** The *DataReader* does not persist the full contents of the data in its historical cache; it only persists an identification (e.g. sequence numbers) of the data the application has processed. This distinction is not meaningful if your application always uses the ‘take’ methods to access your data, since these methods remove the data from the cache at the same time they deliver it to your application. (See 8.4.3.1 Read vs. Take on page 555) However, if your application uses the ‘read’ methods, leaving the data in the *DataReader*'s cache after you've accessed it for the first time, those previously viewed samples will not be restored to the *DataReader*'s cache in the event of a restart.

*Connext DDS* requires a relational database to persist the state of a *DataReader*. This database is accessed using ODBC. See [RTI Connext DDS Core Libraries Database Setup](#) for a list of supported relational databases and setup instructions.

### 13.4.1 Durable Reader State With Protocol Acknowledgment

For each *DataReader* configured to have durable state, *Connext DDS* will create one database table with the following naming convention: RS<32 uuencoding of the reader virtual GUID>. This table will store the last sequence number processed from each virtual GUID. For *DataReaders* on keyed topics requesting instance-ordering (see 7.4.6 PRESENTATION QosPolicy on page 362), this state will be stored per instance per virtual GUID.

**Criteria to consider a sample “processed by the application”**

- For the read/take methods that require calling return_loan(), a sample 's1' with sequence number 's1_seq_num' and virtual GUID ‘vg1’ is considered processed by the application when the *DataReader’s return_loan()* operation is called for sample 's1' or any other sample with the same virtual GUID and a sequence number greater than 's1_seq_num'. For example:
13.4.1 Durable Reader State With Protocol Acknowledgment

```c
retcode = Foo_reader->take(data_seq, info_seq,
    DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
    DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);
if (retcode == DDS_RETCODE_NO_DATA) {
    return;
} else if (retcode != DDS_RETCODE_OK) {
    /* report error */
    return;
}
for (i = 0; i < data_seq.length(); ++i) {
    /* Operate with the data */
}
/* Return the loan */
retcode = Foo_reader->return_loan(data_seq, info_seq);
if (retcode != DDS_RETCODE_OK) {
    /* Report and error */
}
/* At this point the samples contained in data_seq
will be considered as received. If the DataReader
restarts, the samples will not be received again */
```

- For the read/take methods that do not require calling `return_loan()`, a sample 's1' with sequence number 's1_seq_num' and virtual GUID ‘vg1’ will be considered processed after the application reads or takes the sample 's1' or any other sample with the same virtual GUID and with a sequence number greater than 's1_seq_num'. For example:

```c
retcode = Foo_reader->take_next_sample(data, info);
/* At this point the sample contained in data will be
considered as received. All the samples with a sequence
number smaller than the sequence number associated with
data will also be considered as received.
If the DataReader restarts, this sample will not
be received again */
```

If you access the samples in the `DataReader` cache out of order—for example via QueryCondition, specifying an instance state, or reading by instance when the PRESENTATION QoS is not set to INSTANCE_PRESENTATION_QOS—then the samples that have not yet been taken or read by the application may still be considered as "processed by the application".

### 13.4.1.1 Bandwidth Utilization

To optimize network usage, if a `DataReader` configured with durable reader state is restarted and it discovers a `DataWriter` with a virtual GUID ‘vg’, the `DataReader` will ACK all the samples with a sequence number smaller than ‘sn’, where ‘sn’ is the first sequence number that has not been being processed by the application for ‘vg’.

Notice that the previous algorithm can significantly reduce the number of duplicates on the wire. However, it does not suppress them completely in the case of keyed `DataReaders` where the durable state is kept per (instance, virtual GUID). In this case, and assuming that the application has read samples out of order (e.g., by reading different instances), the ACK is sent for the lowest sequence number processed across all
instances and may cause samples already processed to flow on the network again. These redundant samples waste bandwidth, but they will be dropped by the DataReader and not be delivered to the application.

### 13.4.2 Durable Reader State with Application Acknowledgment

This section assumes you are familiar with the concept of Application Acknowledgment as described in 7.3.12 Application Acknowledgment on page 317.

For each DataReader configured to be durable and that uses application acknowledgement (see 7.3.12 Application Acknowledgment on page 317), Connext DDS will create one database table with the following naming convention: RS<32 uuencoding of the reader virtual GUID>. This table will store the list of sequence number intervals that have been acknowledged for each virtual GUID. The size of the column that stores the sequence number intervals is limited to 32767 bytes. If this size is exceeded for a given virtual GUID, the operation that persists the DataReader state into the database will fail.

#### 13.4.2.1 Bandwidth Utilization

To optimize network usage, if a DataReader configured with durable reader state is restarted and it discovers a DataWriter with a virtual GUID ‘vg’, the DataReader will send an APP_ACK message with all the samples that were auto-acknowledged or explicitly acknowledged in previous executions.

Notice that this algorithm can significantly reduce the number of duplicates on the wire. However, it does not suppress them completely since the DataReader may send a NACK and receive some samples from the DataWriter before the DataWriter receives the APP_ACK message.

### 13.4.3 Durable Reader State Use Case

The following use case describes the durable reader state functionality:

1. A DataReader receives two samples with sequence number 1 and 2 published by a DataWriter with virtual GUID 1. The application takes those samples.

   ![Diagram](image)

   1, 2  DataWriter (vg: 1)  1, 2  DataReader (vg: 1)  take 1, 2

2. After the application returns the loan on samples 1 and 2, the DataReader considers them as processed and it persists the state change.
3. The process running the DataReader is stopped.

4. The DataReader is restarted.

Because all the samples with sequence number smaller or equal than 2 were considered received, the reader will not ask for these samples from the DataWriter.

13.4.4 How To Configure a DataReader for Durable Reader State

To configure a DataReader with durable reader state, use the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 associated with DataReaders and DomainParticipants.

A property defined in the DomainParticipant will be applicable to all the DataReaders contained in the participant unless it is overwritten by the DataReaders. Table 13.2 Durable Reader State Properties lists the supported properties.
Table 13.2 Durable Reader State Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.data_reader.state.odbc.dsn</td>
<td>Required. The ODBC DSN (Data Source Name) associated with the database where the DataReader state must be persisted.</td>
</tr>
<tr>
<td>dds.data_reader.state.filter_redundant_samples</td>
<td>To enable durable state, this property must be set to 1. When set to 0, the reader state is not maintained and Connext DDS does not filter duplicate samples that may be coming from the same virtual writer. Default: 1</td>
</tr>
<tr>
<td>dds.data_reader.state.odbc.driver</td>
<td>This property indicates which ODBC driver to load. If the property is not specified, Connext DDS will try to use the standard ODBC driver manager library (UnixOdbc on Linux systems, the Windows ODBC driver manager on Windows systems).</td>
</tr>
<tr>
<td>dds.data_reader.state.odbc.username, dds.data_reader.state.odbc.password</td>
<td>These two properties configure the username and password used to connect to the database. Default: No password or username</td>
</tr>
<tr>
<td>dds.data_reader.state.restore</td>
<td>This property indicates if the persisted DataReader state must be restored or not once the DataReader is restarted. If this property is 0, the previous state will be deleted from the database. If it is 1, the DataReader will restore its previous state from the database content. Default: 1</td>
</tr>
<tr>
<td>dds.data_reader.state.checkpoint_frequency</td>
<td>This property controls how often the reader state is stored into the database. A value of N means store the state once every N samples. A high frequency will provide better performance. However, if the reader is restarted it may receive some duplicate samples. These samples will be filtered by Connext DDS and they will not be propagated to the application. Default: 1</td>
</tr>
<tr>
<td>dds.data_reader.state.persistence_service.request_depth</td>
<td>This property indicates how many of the most recent historical samples the persisted DataReader wants to receive upon start-up. Default: 0</td>
</tr>
</tbody>
</table>

Example (C++ code):

```cpp
/* Get default QoS */
...
retcode = DDSPropertyQosPolicyHelper::add_property{
    readerQos.property,
    "dds.data_reader.state.odbc.dsn",
    "<user DSN>", DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
}
retcode = DDSPropertyQosPolicyHelper::add_property(readerQos.property,
    "dds.data_reader.state.odbc.driver",
    "<ODBC library>", DDS_BOOLEAN_FALSE);
if (retcode != DDS_RETCODE_OK) {
    /* Report error */
```
The data durability feature is an implementation of the OMG DDS Persistence Profile. The 7.5.9 DURABILITY QosPolicy on page 411 allows an application to configure a DataWriter so that the information written by the DataWriter survives beyond the lifetime of the DataWriter.

Connext DDS implements TRANSIENT and PERSISTENT durability using an external service called RTI Persistence Service, available for purchase as a separate RTI product.

Persistence Service receives information from DataWriters configured with TRANSIENT or PERSISTENT durability and makes that information available to late-joining DataReaders—even if the original DataWriter is not running.

The samples published by a DataWriter can be made durable by setting the kind field of the 7.5.9 DURABILITY QosPolicy on page 411 to one of the following values:

- DDS_TRANSIENT_DURABILITY_QOS: Connext DDS will store previously published samples in memory using Persistence Service, which will send the stored data to newly discovered DataReaders.
- DDS_PERSISTENT_DURABILITY_QOS: Connext DDS will store previously published samples in permanent storage, like a disk, using Persistence Service, which will send the stored data to newly discovered DataReaders.

A DataReader can request TRANSIENT or PERSISTENT data by setting the kind field of the corresponding 7.5.9 DURABILITY QosPolicy on page 411. A DataReader requesting PERSISTENT data will not receive data from DataWriters or Persistence Service applications that are configured with TRANSIENT durability.

13.5.1 RTI Persistence Service

Persistence Service is a Connext DDS application that is configured to persist topic data. For each one of the topics that must be persisted for a specific domain, the service will create a DataWriter (known as PRSTDataWriter) and a DataReader (known as PRSTDataReader). The samples received by the PRSTDataReaders will be published by the corresponding PRSTDataWriters to be available for late-joining DataReaders.
For more information on Persistence Service, please see:

- Introduction to RTI Persistence Service (Chapter 45 on page 1180)
- Configuring Persistence Service (Chapter 46 on page 1181)
- Running RTI Persistence Service (Chapter 47 on page 1207)

Persistence Service can be configured to operate in PERSISTENT or TRANSIENT mode:

- **TRANSIENT mode** The PRSTDataReaders and PRSTDataWriters will be created with TRANSIENT durability and Persistence Service will keep the received samples in memory. Samples published by a TRANSIENT DataWriter will survive the DataWriter lifecycle but will not survive the lifecycle of Persistence Service (unless you are running multiple copies).

- **PERSISTENT mode** The PRSTDataWriters and PRSTDataReaders will be created with PERSISTENT durability and Persistence Service will store the received samples in files or in an external relational database. Samples published by a PERSISTENT DataWriter will survive the DataWriter lifecycle as well as any restarts of Persistence Service.

**Peer-to-Peer Communication:**

By default, a PERSISTENT/TRANSIENTDataReader will receive samples directly from the original DataWriter if it is still alive. In this scenario, the DataReader may also receive the same samples from Persistence Service. Duplicates will be discarded at the middleware level. This Peer-To-Peer communication pattern is illustrated in Figure 13.6: Peer-to-Peer Communication on the next page. To use this peer-to-peer communication pattern, set the **direct_communication** field in the 7.5.9 DURABILITY QosPolicy on page 411 to TRUE. A PERSISTENT/TRANSIENT DataReader will receive information directly from PERSISTENT/TRANSIENT DataWriters.
Figure 13.6: Peer-to-Peer Communication

![Diagram of Peer-to-Peer Communication]

**Relay Communication**

A PERSISTENT/TRANSIENT DataReader may also be configured to not receive samples from the original DataWriter. In this case the traffic is relayed by Persistence Service. This ‘relay communication’ pattern is illustrated in Figure 13.7: Relay Communication below. To use relay communication, set the direct communication field in the 7.5.9 DURABILITY QosPolicy on page 411 to FALSE. A PERSISTENT/TRANSIENT DataReader will receive all the information from Persistence Service.

Figure 13.7: Relay Communication

![Diagram of Relay Communication]
Chapter 14 Guaranteed Delivery of Data

14.1 Introduction

Some application scenarios need to ensure that the information produced by certain producers is delivered to all the intended consumers. This chapter describes the mechanisms available in Connext DDS to guarantee the delivery of information from producers to consumers such that the delivery is robust to many kinds of failures in the infrastructure, deployment, and even the producing/consuming applications themselves.

Guaranteed information delivery is not the same as protocol-level reliability (described in Reliable Communications (Chapter 11 on page 688)) or information durability (described in Mechanisms for Achieving Information Durability and Persistence (Chapter 13 on page 732)). Guaranteed information delivery is an end-to-end application-level QoS, whereas the others are middleware-level QoS. There are significant differences between these two:

- With protocol-level reliability alone, the producing application knows that the information is received by the protocol layer on the consuming side. However the producing application cannot be certain that the consuming application read that information or was able to successfully understand and process it. The information could arrive in the consumer’s protocol stack and be placed in the DataReader cache but the consuming application could either crash before it reads it from the cache, not read its cache, or read the cache using queries or conditions that prevent that particular DDS data sample from being accessed. Furthermore, the consuming application could access the DDS sample, but not be able to interpret its meaning or process it in the intended way.

- With information durability alone, there is no way to specify or characterize the intended consumers of the information. Therefore the infrastructure has no way to know when the information has been consumed by all the intended recipients. The information may be persisted such that it is not lost and is available to future applications, but the infrastructure and producing applications have no way to know that all the intended consumers have joined the system, received the information, and processed it successfully.
14.1.1 Identifying the Required Consumers of Information

The guaranteed data-delivery mechanism provided in Connext DDS overcomes the limitations described above by providing the following features:

- **Required subscriptions.** This feature provides a way to configure, identify and detect the applications that are intended to consume the information. See 7.3.13 Required Subscriptions on page 323.

- **Application-level acknowledgments.** This feature provides the means ensure that the information was successfully processed by the application-layer in a consumer application. See 7.3.12 Application Acknowledgment on page 317.

- **Durable subscriptions.** This feature leverages the RTI Persistence Service to persist DDS DDS samples intended for the required subscriptions such that they are delivered even if the originating application is not available. See 46.9 Configuring Durable Subscriptions in Persistence Service on page 1200.

These features used in combination with the mechanisms provided for Information Durability and Persistence (see Mechanisms for Achieving Information Durability and Persistence (Chapter 13 on page 732)) enable the creation of applications where the information delivery is guaranteed despite application and infrastructure failures. 14.2 Scenarios on page 757 describes various guaranteed-delivery scenarios and how to configure the applications to achieve them.

When implementing an application that needs guaranteed data delivery, we have to consider three key aspects:

<table>
<thead>
<tr>
<th>Key Aspects to Consider</th>
<th>Related Features and QoS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifying the required consumers of information</td>
<td>Required subscriptions</td>
</tr>
<tr>
<td></td>
<td>Durable subscriptions</td>
</tr>
<tr>
<td></td>
<td>EntityName QoS policy</td>
</tr>
<tr>
<td></td>
<td>Availability QoS policy</td>
</tr>
<tr>
<td>Ensuring the intended consumer applications process the data successfully</td>
<td>Application-level acknowledgment</td>
</tr>
<tr>
<td></td>
<td>Acknowledgment by a quorum of required and durable subscriptions</td>
</tr>
<tr>
<td></td>
<td>Reliability QoS policy (acknowledgment mode)</td>
</tr>
<tr>
<td></td>
<td>Availability QoS policy</td>
</tr>
<tr>
<td>Ensuring information is available to late joining applications</td>
<td>Persistence Service</td>
</tr>
<tr>
<td></td>
<td>Durable Subscriptions</td>
</tr>
<tr>
<td></td>
<td>Durability QoS</td>
</tr>
<tr>
<td></td>
<td>Durable Writer History</td>
</tr>
</tbody>
</table>

**14.1.1 Identifying the Required Consumers of Information**

The first step towards ensuring that information is processed by the intended consumers is the ability to specify and recognize those intended consumers. This is done using the *required subscriptions* feature (7.3.13...
14.1.1 Identifying the Required Consumers of Information

Required Subscriptions on page 323) configured via the 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418 and 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370).

Connext DDS DataReader entities (as well as DataWriter and DomainParticipant entities) can have a name and a role_name. These names are configured using the 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418, which is propagated via DDS discovery and is available as part of the builtin-topic data for the Entity (see Built-In Topics (Chapter 18 on page 831)).

The DDS DomainParticipant, DataReader and DataWriter entities created by RTI-provided applications and services, specifically services such as RTI Persistence Service, automatically configure the ENTITY_NAME QoS policy according to their function. For example the DataReaders created by RTI Persistence Service have their role_name set to “PERSISTENCE_SERVICE”.

Unless explicitly set by the user, the DomainParticipant, DataReader and DataWriter entities created by end-user applications have their name and role_name set to NULL. However applications may modify this using the 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.

Connext DDS uses the role_name of DataReaders to identify the consumer’s logical function. For this reason Connext DDS’s required subscriptions feature relies on the role_name to identify intended consumers of information. The use of the DataReader’s role_name instead of the name is intentional. From the point of view of the information producer, the important thing is not the concrete DataReader (identified by its name, for example, “Logger123”) but rather its logical function in the system (identified by its role_name, for example “LoggingService”).

A DataWriter that needs to ensure its information is delivered to all the intended consumers uses the 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370 to configure the role names of the consumers that must receive the information.

The AVAILABILITY QoS Policy set on a DataWriter lets an application configure the required consumers of the data produced by the DataWriter. The required consumers are specified in the required_matched_endpoint_groups attribute within the AVAILABILITY QoS Policy. This attribute is a sequence of DDS EndpointGroup structures. Each EndpointGroup represents a required information consumer characterized by the consumer’s role_name and quorum_count. The role_name identifies a logical consumer; the quorum_count specifies the minimum number of consumers with that role_name that must acknowledge the DDS sample before the DataWriter can consider it delivered to that required consumer.

For example, an application that wants to ensure data written by a DataWriter is delivered to at least two Logging Services and one Display Service would configure the DataWriter’s AVAILABILITY QoS Policy with a required_matched_endpoint_groups consisting of two elements. The first element would specify a required consumer with the role_name “LoggingService” and a quorum_count of 2. The second element would specify a required consumer with the role_name “DisplayService” and a quorum_count of 1. Furthermore, the application would set the logging service DataReader ENTITY_NAME policy to have a role_name of “LoggingService” and similarly the display service DataReader ENTITY_NAME policy to have the role_name of “DisplayService.”
A DataWriter that has been configured with an AVAILABILITY QoS policy will not remove DDS samples from the DataWriter cache until they have been “delivered” to both the already-discovered DataReaders and the minimum number (quorum_count) of DataReaders specified for each role. In particular, DDS samples will be retained by the DataWriter if the quorum_count of matched DataReaders with a particular role_name have not been discovered yet.

We used the word “delivered” in quotes above because the level of assurance a DataWriter has that a particular DDS sample has been delivered depends on the setting of the 7.5.21 RELIABILITY QosPolicy on page 447. We discuss this next in 14.1.2 Ensuring Consumer Applications Process the Data Successfully below.

14.1.2 Ensuring Consumer Applications Process the Data Successfully

14.1.1 Identifying the Required Consumers of Information on page 753 described mechanisms by which an application could configure who the required consumers of information are. This section is about the criteria, mechanisms, and assurance provided by Connext DDS to ensure consumers have the information delivered to them and process it in a successful manner.

RTI provides four levels of information delivery guarantee. You can set your desired level using the 7.5.21 RELIABILITY QosPolicy on page 447. The levels are:

- **Best-effort, relying only on the underlying transport** The DataWriter considers the DDS sample delivered/acknowledged as soon as it is given to the transport to send to the DataReader’s destination. Therefore, the only guarantee is the one provided by the underlying transport itself. Note that even if the underlying transport is reliable (e.g., shared memory or TCP) the reliability is limited to the transport-level buffers. There is no guarantee that the DDS sample will arrive to the DataReader cache because after the transport delivers to the DataReader’s transport buffers, it is possible for the DDS sample to be dropped because it exceeds a resource limit, fails to deserialize properly, the receiving application crashes, etc.

- **Reliable with protocol acknowledgment** The DDS-RTPS reliability protocol used by Connext DDS provides acknowledgment at the RTPS protocol level: a DataReader will acknowledge it has deserialized the DDS sample correctly and stored it in the DataReader’s cache. However, there is no guarantee the application actually processed the DDS sample. The application might crash before processing the DDS sample, or it might simply fail to read it from the cache.

- **Reliable with Application Acknowledgment (Auto)** Application Acknowledgment in Auto mode causes Connext DDS to send an additional application-level acknowledgment (above and beyond the RTPS protocol level acknowledgment) after the consuming application has read the DDS sample from the DataReader cache and the application has subsequently called the DataReader’s return_loan() operation (see 8.4.2 Loaning and Returning Data and SampleInfo Sequences on page 553) for that DDS sample. This mode guarantees that the application has fully read the DDS sample all the way until it indicates it is done with it. However it does not provide a guarantee that the application was able to successfully interpret or process the DDS sample. For example, the DDS
sample could be a command to execute a certain action and the application may read the DDS sample and not understand the command or may not be able to execute the action.

- **Reliable with Application Acknowledgment (Explicit)** Application Acknowledgment in Explicit mode causes Connext DDS to send an application-level acknowledgment only after the consuming application has read the DDS sample from the DataReader cache and subsequently called the DataReader’s `acknowledge_sample()` operation (see 8.4.4 Acknowledging DDS Samples on page 563) for that DDS sample. This mode guarantees that the application has fully read the DDS sample and completed operating on it as indicated by explicitly calling `acknowledge_sample()`. In contrast with the Auto mode described above, the application can delay the acknowledgment of the DDS sample beyond the time it holds onto the data buffers, allowing it to be process in a more flexible manner. Similar to the Auto mode, it does not provide a guarantee that the application was able to successfully interpret or process the DDS sample. For example, the DDS sample could be a command to execute a certain action and the application may read the DDS sample and not understand the command or may not be able to execute the action. Applications that need guarantees that the data was successfully processed and interpreted should use a request-reply interaction (see Part 4: Request-Reply and RPC Communication Patterns on page 999).

### 14.1.3 Ensuring Information is Available to Late-Joining Applications

The third aspect of guaranteed data delivery addresses situations where the application needs to ensure that the information produced by a particular DataWriter is available to DataReaders that join the system after the data was produced. The need for data delivery may even extend beyond the lifetime of the producing application; that is, it may be required that the information is delivered to applications that join the system after the producing application has left the system.

Connext DDS provides four mechanisms to handle these scenarios:

- **The DDS Durability QoS Policy.** The 7.5.9 DURABILITY QoS Policy on page 411 specifies whether DDS samples should be available to late joiners. The policy is set on the DataWriter and the DataReader and supports four kinds: VOLATILE, TRANSIENT_LOCAL, TRANSIENT, or PERSISTENT. If the DataWriter’s Durability QoS policy is set to VOLATILE kind, the DataWriter’s DDS samples will not be made available to any late joiners. If the DataWriter’s policy kind is set to TRANSIENT_LOCAL, TRANSIENT, or PERSISTENT, the DDS samples will be made available for late-joining DataReaders who also set their DURABILITY QoS policy kind to something other than VOLATILE.

- **Durable Writer History.** A DataWriter configured with a DURABILITY QoS policy kind other than VOLATILE keeps its data in a local cache so that it is available when the late-joining application appears. The data is maintained in the DataWriter’s cache until it is considered to be no longer needed. The precise criteria depends on the configuration of additional QoS policies such as 7.5.14 LIFESPAN QoS Policy on page 425, 7.5.12 HISTORY QoSPolicy on page 420, 7.5.22 RESOURCE_LIMITS QoSPolicy on page 451, etc. For the purposes of guaranteeing information delivery it is important to note that the DataWriter’s cache can be configured to be a memory cache
or a durable (disk-based) cache. A memory cache will not survive an application restart. However, a durable (disk-based) cache can survive the restart of the producing application. The use of a durable writer history, including the use of an external ODBC database as a cache, is described in 13.3 Durable Writer History on page 739.

- **RTI Persistence Service.** This service allows the information produced by a DataWriter to survive beyond the lifetime of the producing application. Persistence Service is an stand-alone application that runs on many supported platforms. This service complies with the Persistent Profile of the OMG DDS specification. The service uses DDS to subscribe to the DataWriters that specify a 7.5.9 DURABILITY QosPolicy on page 411 kind of TRANSIENT or PERSISTENT. Persistence Service receives the data from those DataWriters, stores the data in its internal caches, and makes the data available via DataReaders (which are automatically created by Persistence Service) to late-joining DataReaders that specify a Durability kind of TRANSIENT or PERSISTENT. Persistence Service can operate as a relay for the information from the original writer, preserving the source_timestamp of the data, as well as the original DDS sample virtual writer GUID (see 13.5.1 RTI Persistence Service on page 749). In addition, you can configure Persistence Service itself to use a memory-based cache or a durable (disk-based or database-based) cache. See 46.6 Configuring Persistent Storage on page 1189. Configuration of redundant and load-balanced persistence services is also supported.

- **Durable Subscriptions.** This is a Persistence Service configuration setting that allows configuration of the required subscriptions (14.1.1 Identifying the Required Consumers of Information on page 753) for the data stored by Persistence Service (7.3.14 Managing Instances (Working with Keyed Data Types) on page 325). Configuring required subscriptions for Persistence Service ensures that the service will store the DDS samples until they have been delivered to the configured number (quorum_count) of DataReaders that have each of the specified roles.

## 14.2 Scenarios

In each of the scenarios below, we assume both the DataWriter andDataReader are configured for strict reliability (RELIABLE ReliabilityQosPolicyKind and KEEP_ALL HistoryQosPolicyKind, see 11.3.3 Controlling Queue Depth with the History QosPolicy on page 703). As a result, when the DataWriter’s cache is full of unacknowledged DDS samples, the write() operation will block until DDS samples are acknowledged by all the intended consumers.

### 14.2.1 Scenario 1: Guaranteed Delivery to a-priori Known Subscribers

A common use case is to guarantee delivery to a set of known subscribers. These subscribers may be already running and have been discovered, they may be temporarily non-responsive, or it could be that some of those subscribers are still not present in the system. See Figure 14.1: Guaranteed Delivery Scenario 1 on the next page.

To guarantee delivery, the list of required subscribers should be configured using the 7.5.1 AVAILABILITY QosPolicy (DDS Extension) on page 370 on the DataWriters to specify the role_name
14.2.1 Scenario 1: Guaranteed Delivery to a-priori Known Subscribers

and quorum_count for each required subscription. Similarly the 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418 should be used on the DataReaders to specify their role_name. In addition we use 7.3.12 Application Acknowledgment on page 317 to guarantee the DDS sample was delivered and processed by the DataReader.

Figure 14.1: Guaranteed Delivery Scenario 1

The DataWriter's and DataReader's RELIABILITY QoS Policy can be configured for either AUTO or EXPLICIT application acknowledgment kind. As the DataWriter publishes the DDS sample, it will await
acknowledgment from the DataReader (through the protocol-level acknowledgment) and from the subscriber application (though the additional application-level acknowledgment). The DataWriter will only consider the DDS sample acknowledged when it has been acknowledged by all discovered active DataReaders and also by the quorum_count of each required subscription.

In this specific scenario, DataReader #1 is configured for EXPLICIT application acknowledgment. After reading and processing the DDS sample, the subscribing application calls acknowledge_sample() or acknowledge_all() (see 8.4.4 Acknowledging DDS Samples on page 563). As a result, Connext DDS will send an application-level acknowledgment to the DataWriter, which will in its turn confirm the acknowledgment.

If the DDS sample was lost in transit, the reliability protocol will repair the DDS sample. Since it has not been acknowledged, it remains available in the writer’s queue to be automatically resent by Connext DDS. The DDS sample will remain available until acknowledged by the application. If the subscribing application crashes while processing the DDS sample and restarts, Connext DDS will repair the unacknowledged DDS sample. DDS samples which already been processed and acknowledged will not be resent.

In this scenario, DataReader #2 may be a late joiner. When it starts up, because it is configured with TRANSIENT_LOCAL Durability, the reliability protocol will re-send the DDS samples previously sent by the writer. These DDS samples were considered unacknowledged by the DataWriter because they had not been confirmed yet by the required subscription (identified by its role_name: ‘logger’).

DataReader #2 does not explicitly acknowledge the DDS samples it reads. It is configured to use AUTO application acknowledgment, which will automatically acknowledge DDS samples that have been read or taken after the application calls the DataReader return_loan operation.

This configuration works well for situations where the DataReader may not be immediately available or may restart. However, this configuration does not provide any guarantee if the DataWriter restarts. When the DataWriter restarts, DDS samples previously unacknowledged are lost and will no longer be available to any late joining DataReaders.

### 14.2.2 Scenario 2: Surviving a Writer Restart when Delivering DDS Samples to a priori Known Subscribers

Scenario 1 describes a use case where DDS samples are delivered to a list of a priori known subscribers. In that scenario, Connext DDS will deliver DDS samples to the late-joining or restarting subscriber. However, if the producer is re-started the DDS samples it had written will no longer be available to future subscribers.

To handle a situation where the producing application is restarted, we will use the 13.3 Durable Writer History on page 739 feature. See Figure 14.2: Guaranteed Delivery Scenario 2 on the next page.

A DataWriter can be configured to maintain its data and state in durable storage. This configuration is done using the PROPERTY QoS policy as described in 13.3.2 How To Configure Durable Writer
14.2.3 Scenario 3: Delivery Guaranteed by Persistence Service (Store and Forward) to a priori Known Subscribers

Previous scenarios illustrated that using the DURABILITY, RELIABILITY, and AVAILABILITY QoS policies we can ensure that as long as the DataWriter is present in the system, DDS samples written by a DataWriter will be delivered to the intended consumers. The use of the durable writer history in the previous scenario extended this guarantee even in the presence of a restart of the application writing the data.

This scenario addresses the situation where the originating application that produced the data is no longer available. For example, the network could have become partitioned, the application could have crashed and not have been restarted, etc.
14.2.3 Scenario 3: Delivery Guaranteed by Persistence Service (Store and Forward) to a priori Known

In order to deliver data to applications that appear after the producing application is no longer available on the network it is necessary to have another service that stores those DDS samples and delivers them. This is the purpose of the RTI Persistence Service.

Persistence Service can be configured to automatically discover DataWriters that specify a DURABILITY QoS with kind TRANSIENT or PERSISTENT and automatically create pairs (DataReader, DataWriter) that receive and store that information (see Introduction to RTI Persistence Service (Chapter 45 on page 1180)).

All DataReaders created by the RTI Persistence Service have the ENTITY_QOS policy set with the role_name of “PERSISTENCE_SERVICE”. This allows an application to specify Persistence Service as one of the required subscriptions for its DataWriters.

In this third scenario, we take advantage of this capability to configure the DataWriter to have the RTI Persistence Service as a required subscription. See Figure 14.3: Guaranteed Delivery Scenario 3 below.

![Figure 14.3: Guaranteed Delivery Scenario 3](image)

The RTI Persistence Service can also have its DataWriters configured with required subscriptions. This feature is known as Persistence Service “durable subscriptions”. DataReader #1 is pre configured in Persistence Service as a Durable Subscription. (Alternatively, DataReader #1 could have registered itself dynamically as Durable Subscription using the DomainParticipant register_durable_subscription() operation).
14.2.3 Scenario 3: Delivery Guaranteed by Persistence Service (Store and Forward) to a priori Known

We also configure the RELIBILITY QoS policy setting of the AcknowledgmentKind to APPLICATION_AUTO_ACKNOWLEDGMENT_MODE in order to ensure DDS samples are stored in the Persistence Service and properly processed on the consuming application prior to them being removed from the DataWriter cache.

With this configuration in place the DataWriter will deliver DDS samples to the DataReader and to the Persistence Service reliably and wait for the Application Acknowledgment from both. Delivery of DDS samples to DataReader #1 and the Persistence Service occurs concurrently. The Persistence Service in turn takes responsibility to deliver the DDS samples to the configured “logger” durable subscription. If the original publisher is no longer available, DDS samples can still be delivered by the Persistence Service. to DataReader #1 and any other late-joining DataReaders.

When DataReader #1 acknowledges the DDS sample through an application-acknowledgment message, both the original DataWriter and Persistence Service will receive the application-acknowledgment. Connext DDS takes advantage of this to reduce or eliminate delivery if duplicate DDS samples, that is, the Persistence Service can notice that DataReader #1 has acknowledged a DDS sample and refrain from separately sending the same DDS sample to DataReader #1.

14.2.3.1 Variation: Using Redundant Persistence Services

Using a single Persistence Service to guarantee delivery can still raise concerns about having the Persistence Service as a single point of failure. To provide a level of added redundancy, the publisher may be configured to await acknowledgment from a quorum of multiple persistence services (role_name remains PERSISTENCE). Using this configuration we can achieve higher levels of redundancy

Figure 14.4: Guaranteed Delivery Scenario 3 with Redundant Persistence Service

- **ReliabilityQoS**: Reliable kind
- **History kind**: KEEP_ALL
- **Durability**: PERSISTENT
- **AUTO Acknowledgment kind**
- **Required_matched_endpoint_groups**: PERSISTENCE
- **Quorum**: 2

![Diagram of Guaranteed Delivery Scenario 3 with Redundant Persistence Service]
The RTI Persistence Services will automatically share information to keep each other synchronized. This includes both the data and also the information on the durable subscriptions. That is, when a Persistence Service discovers a durable subscription, information about durable subscriptions is automatically replicated and synchronized among persistence services (CITE: New section to be written in Persistence Service Chapter).

14.2.3.2 Variation: Using Load-Balanced Persistent Services

The Persistence Service will store DDS samples on behalf of many DataWriters and, depending on the configuration, it might write those DDS samples to a database or to disk. For this reason the Persistence Service may become a bottleneck in systems with high durable DDS sample throughput.

It is possible to run multiple instances of the Persistence Service in a manner where each is only responsible for the guaranteed delivery of certain subset of the durable data being published. These Persistence Service can also be run different computers and in this manner achieve much higher throughput. For example, depending on the hardware, using typical hard-drives a single a Persistence Service may be able to store only 30000 DDS samples per second. By running 10 persistence services in 10 different computers we would be able to handle storing 10 times that system-wide, that is, 300000 DDS samples per second.

The data to be persisted can be partitioned among the persistence services by specifying different Topics to be persisted by each Persistence Service. If a single Topic has more data that can be handled y a single Persistence Service it is also possible to specify a content-filter so that only the data within that Topic that matches the filter will be stored by the Persistence Service. For example assume the Topic being persisted has an member named “x” of type float. It is possible to configure two Persistence Services one with the filter “x>10”, and the other “x <=10”, such that each only stores a subject of the data published on the Topic. See also: 46.9 Configuring Durable Subscriptions in Persistence Service on page 1200.
Chapter 15 Discovery

This section discusses how Connext DDS objects on different nodes find out about each other using the default Simple Discovery Protocol (SDP). It describes the sequence of messages that are passed between Connext DDS on the sending and receiving sides.

This section includes:

- 15.1 What is Discovery? on the next page
- 15.2 Configuring the Peers List Used in Discovery on page 767
- 15.3 Discovery Implementation on page 773
- 15.4 Debugging Discovery on page 789
- 15.5 Ports Used for Discovery on page 791

The discovery process occurs automatically, so you do not have to implement any special code. We recommend that all users read 15.1 What is Discovery? on the next page and 15.2 Configuring the Peers List Used in Discovery on page 767. The remaining sections contain advanced material for those who have a particular need to understand what is happening ‘under the hood.’ This information can help you debug a system in which objects are not communicating.

You may also be interested in reading Transport Plugins (Chapter 16 on page 796), as well as learning about these QosPolicies:

- 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459
- 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665
- 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460
- 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) on page 591
15.1 What is Discovery?

Discovery is the behind-the-scenes way in which Connext DDS objects (DomainParticipants, DataWriters, and DataReaders) on different nodes find out about each other. Each DomainParticipant maintains a database of information about all the active DataReaders and DataWriters that are in the same DDS domain. This database is what makes it possible for DataWriters and DataReaders to communicate. To create and refresh the database, each application follows a common discovery process.

This chapter describes the default discovery mechanism known as the Simple Discovery Protocol, which includes two phases: 15.1.1 Simple Participant Discovery below and 15.1.2 Simple Endpoint Discovery on the next page.

The goal of these two phases is to build, for each DomainParticipant, a complete picture of all the entities that belong to the remote participants that are in its peers list. The peers list is the list of nodes with which a participant may communicate. It starts out the same as the initial_peers list that you configure in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640. If the accept_unknown_peers flag in that same QosPolicy is TRUE, then other nodes may also be added as they are discovered; if it is FALSE, then the peers list will match the initial_peers list, plus any peers added using the DomainParticipant’s add_peer() operation.

15.1.1 Simple Participant Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Participant Discovery Protocol (SPDP).

During the Participant Discovery phase, DomainParticipants learn about each other. The DomainParticipant’s details are communicated to all other DomainParticipants in the same DDS domain by sending participant declaration messages, also known as participant DATA submessages or participant announcements. The details include the DomainParticipant’s unique identifying key (GUID or Globally Unique ID described below), transport locators (addresses and port numbers), and QoS. These messages are sent on a periodic basis using best-effort communication.

Participant DATAs are sent periodically to maintain the liveliness of the DomainParticipant. They are also used to communicate changes in the DomainParticipant’s QoS. Only changes to QosPolicies that are part of the DomainParticipant’s built-in data (namely, the 7.5.30 USER_DATA QosPolicy on page 465) need to be propagated.

When receiving remote participant discovery information, Connext DDS determines if the local participant matches the remote one. A ‘match’ between the local and remote participant occurs only if the local and remote participant have the same Domain ID and Domain Tag (see 9.3.4 Choosing a Domain ID and Creating Multiple DDS Domains on page 618 and 9.3.5 Choosing a Domain Tag on page 619). This matching process occurs as soon as the local participant receives discovery information from the remote one. If there is no match, the discovery DATA is ignored, resulting in the remote participant (and all its associated entities) not being discovered.
When a DomainParticipant is deleted, a participant DATA (delete) submessage with the DomainParticipant's identifying GUID is sent.

The GUID is a unique reference to an entity. It is composed of a GUID prefix and an Entity ID. By default, the GUID prefix is calculated from the IP address and the process ID. (For more on how the GUID is calculated, see 9.5.9.3 Controlling How the GUID is Set (rtps_auto_id_kind) on page 673.) The IP address and process ID are stored in the DomainParticipant’s 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670. The entityID is set by Connext DDS (you may be able to change it in a future version).

Once a pair of participants have discovered each other, they can move on to the Endpoint Discovery phase, which is how DataWriters and DataReaders find each other.

You may replace Simple Participant Discovery with Limited Bandwidth Participant Discovery (LBPD) or other discovery protocols. See builtin_discovery_plugins in the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644 for more information.

15.1.2 Simple Endpoint Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Endpoint Discovery Protocol (SEDP).

During the Endpoint Discovery phase, Connext DDS matches DataWriters and DataReaders. Information (GUID, QoS, etc.) about your application’s DataReaders and DataWriters is exchanged by sending publication/subscription declarations in DATA messages that we will refer to as publication DATAs and subscription DATAs. The Endpoint Discovery phase uses reliable communication.

As described in 15.3 Discovery Implementation on page 773, these declaration or DATA messages are exchanged until each DomainParticipant has a complete database of information about the participants in its peers list and their entities. Then the discovery process is complete and the system switches to a steady state. During steady state, participant DATAs are still sent periodically to maintain the liveness status of participants. They may also be sent to communicate QoS changes or the deletion of a DomainParticipant.

When a remote DataWriter/DataReader is discovered, Connext DDS determines if the local application has a matching DataReader/DataWriter. A ‘match’ between the local and remote entities occurs only if the DataReader and DataWriter have the same Topic, same data type, and compatible QoS Policies (which includes having the same partition name string, see 7.4.5 PARTITION QosPolicy on page 356). Furthermore, if the DomainParticipant has been set up to ignore certain DataWriters/DataReaders, those entities will not be considered during the matching process. See 18.4.2 Ignoring Publications and Subscriptions on page 843 for more on ignoring specific publications and subscriptions.

This ‘matching’ process occurs as soon as a remote entity is discovered, even if the entire database is not yet complete: that is, the application may still be discovering other remote entities.

A DataReader and DataWriter can only communicate with each other if each one’s application has hooked up its local entity with the matching remote entity. That is, both sides must agree to the connection.
15.3 Discovery Implementation on page 773 describes the details about the discovery process.

You may replace Simple Endpoint Discovery with Limited Bandwidth Endpoint Discovery (LBED) or other discovery protocols. See builtin_discovery_plugins in the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644 for more information.

15.2 Configuring the Peers List Used in Discovery

As part of the participant phase of the discovery process, Connext DDS will announce itself within the DDS domain. Connext DDS will try to contact all possible participants in the ‘initial peers list,’ specified in the DomainParticipant’s 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640. Note, however, it is not known if there are actually Connext DDS applications running on the hosts in the initial peers list. The initial peers list may include both unicast and multicast peer locators.

After startup, you can add to the ‘peers list’ with the add_peer() operation (see 9.5.2.3 Adding and Removing Peers List Entries on page 641). The ‘peers list’ may also grow as peers are automatically discovered (if acceptUnknownPeers is TRUE, see 9.5.2.6 Controlling Acceptance of Unknown Peers on page 642).

When you call get_default_participant_qos() for a DomainParticipantFactory, the values used for the DiscoveryQosPolicy’s initial_peers and multicast_receive_addresses may come from the following:

- A file named NDDS_DISCOVERY_PEERS, which is formatted as described in 15.2.3 NDDS_DISCOVERY_PEERS File Format on page 772. The file must be in your application’s current working directory.
- An environment variable named NDDS_DISCOVERY_PEERS, defined as a comma-separated list of peer descriptors (see 15.2.2 NDDS_DISCOVERY_PEERS Environment Variable Format on page 771).
- The value specified in the default XML QoS profile (see 19.4 Tags for Configuring QoS with XML on page 874).

If NDDS_DISCOVERY_PEERS (file or environment variable) does not contain a multicast address, then multicast_receive_addresses is cleared and the RTI discovery process will not listen for discovery messages via multicast.

If NDDS_DISCOVERY_PEERS (file or environment variable) contains one or more multicast addresses, the addresses are stored in multicast_receive_addresses, starting at element 0. They will be stored in the order in which they appear in NDDS_DISCOVERY_PEERS.

Note: Setting initial_peers in the default XML QoS Profile does not modify the value of multicast_receive_address.
15.2 Configuring the Peers List Used in Discovery

If both the file and environment variable are found, the file takes precedence and the environment variable will be ignored.\(^1\) The settings in the default XML QoS Profile take precedence over the file and environment variable. In the absence of a file, environment variable, or default XML QoS profile values, Connext DDS will use a default value. See the API Reference HTML documentation for details (in the section on the DISCOVERY QosPolicy).

If initial peers are specified in both the currently loaded QoS XML profile and in the NDDS_DISCOVERY_PEERS file, the values in the profile take precedence.

The file, environment variable, and default XML QoS Profile make it easy to reconfigure which nodes will take part in the discovery process—without recompiling your application.

The file, environment variable, and default XML QoS Profile are the possible sources for the default initial peers list. You can, of course, explicitly set the initial list by changing the values in the QoS provided to the DomainParticipantFactory's `create_participant()` operation, or by adding to the list after startup with the `DomainParticipant`'s `add_peer()` operation (see 9.5.2.3 Adding and Removing Peers List Entries on page 641).

**If you set NDDS_DISCOVERY_PEERS and You Want to Communicate over Shared Memory:**

Suppose you want to communicate with other Connext DDS applications on the same host and you are explicitly setting NDDS_DISCOVERY_PEERS (generally in order to use unicast discovery with applications on other hosts).

If the local host platform does not support the shared memory transport, then you can include the name of the local host in the NDDS_DISCOVERY_PEERS list. (To check if your platform supports shared memory, see the RTI Connext DDS Core Libraries Platform Notes.)

If the local host platform supports the shared memory transport, then you must do one of the following:

- Include "shmem://" in the NDDS_DISCOVERY_PEERS list. This will cause shared memory to be used for discovery and data traffic for applications on the same host.

or:

- Include the name of the local host in the NDDS_DISCOVERY_PEERS list, and disable the shared memory transport in the 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665 of the DomainParticipant. This will cause UDP loopback to be used for discovery and data traffic for applications on the same host.

\(^1\)This is true even if the file is empty.
15.2.1 Peer Descriptor Format

A peer descriptor string specifies a range of participants at a given locator. Peer descriptor strings are used in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640 initial_peers field (see 9.5.2.2 Setting the 'Initial Peers’ List on page 640) and the DomainParticipant’s add-peer() and remove_peer() operations (see 9.5.2.3 Adding and Removing Peers List Entries on page 641).

The anatomy of a peer descriptor is illustrated in Figure 15.1: Example Peer Descriptor Address Strings below using a special "StarFabric" transport example.

Figure 15.1: Example Peer Descriptor Address Strings

A peer descriptor consists of:

- [optional] A participant ID limit. If a simple integer is specified, it indicates the maximum participant ID to be contacted by the Connext DDS discovery mechanism at the given locator. If that integer is enclosed in square brackets (e.g., [2]), then only that Participant ID will be used. You can also specify a range in the form of [a-b]; in this case only the Participant IDs in that specific range are contacted. If omitted, a default value of 4 is implied and participant IDs 0, 1, 2, 3, and 4 will be contacted.

- A locator, as described in 15.2.1.1 Locator Format on the next page.

These are separated by the '@' character. The separator may be omitted if a participant ID limit is not explicitly specified.
The "participant ID limit" only applies to unicast locators; it is ignored for multicast locators (and therefore should be omitted for multicast peer descriptors).

### 15.2.1.1 Locator Format

A locator string specifies a transport and an address in string format. Locators are used to form peer descriptors. A locator is equivalent to a peer descriptor with the default participant ID limit (4).

A locator consists of:

- [optional] Transport name (alias or class). This identifies the set of transport plug-ins (transport aliases) that may be used to parse the address portion of the locator. Note that a transport class name is an implicit alias used to refer to all the transport plug-in instances of that class.

- [optional] An address, as described in 15.2.1.2 Address Format below.

These are separated by the "://" string. The separator is specified if and only if a transport name is specified.

If a transport name is specified, the address may be omitted; in that case all the unicast addresses (across all transport plug-in instances) associated with the transport class are implied. Thus, a locator string may specify several addresses.

If an address is specified, the transport name and the separator string may be omitted; in that case all the available transport plug-ins for the Entity may be used to parse the address string.

The transport names for the built-in transport plug-ins are:

- shmem - Shared Memory Transport
- udpv4 - UDPv4 Transport
- udpv6 - UDPv6 Transport

### 15.2.1.2 Address Format

An address string specifies a transport-independent network address that qualifies a transport-dependent address string. Addresses are used to form locators. Addresses are also used in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640 multicast_receive_addresses and the DDS_TransportMulticastSettings_t::receive_address fields. An address is equivalent to a locator in which the transport name and separator are omitted.

An address consists of:

- [optional] A network address in IPv4 or IPv6 string notation. If omitted, the network address of the transport is implied.
[optional] A transport address, which is a string that is passed to the transport for processing. The transport maps this string into NDDS_Transport_Property_t::address_bit_count bits. If omitted, the network address is used as the fully qualified address. The transport plugin sets the value for NDDS_Transport_Property_t::address_bit_count bits.

The network and transport addressed are separated by the '#' character. If a separator is specified, it must be followed by a non-empty string that is passed to the transport plug-in. If the separator is omitted, it is treated as a transport address with an implicit network address (of the transport plugin). The implicit network address is the address used when registering the transport: e.g., the UDPv4 implicit network address is 0.0.0.0.0.0.0.0.0.0.0.0.0.0.

The bits resulting from the transport address string are prepended with the network address. The least significant NDDS_Transport_Property_t::address_bit_count bits of the network address are ignored.

## 15.2.2 NDDS_DISCOVERY_PEERS Environment Variable Format

You can set the default value for the initial peers list in an environment variable named NDDS_DISCOVERY_PEERS. Multiple peer descriptor entries must be separated by commas. Table 15.1 NDDS_DISCOVERY_PEERS Environment Variable Examples shows some examples. The examples use an implied maximum participant ID of 4 unless otherwise noted. (If you need instructions on how to set environment variables, see Set Up Environment Variables (rtisetenv), in "Hands-On 1" of Introduction to Publish/Subscribe, in the RTI Connext DDS Getting Started Guide.)

### Table 15.1 NDDS_DISCOVERY_PEERS Environment Variable Examples

<table>
<thead>
<tr>
<th>NDDS_DISCOVERY_PEERS</th>
<th>Description of Host(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>239.255.0.1</td>
<td>multicast</td>
</tr>
<tr>
<td>localhost</td>
<td>localhost</td>
</tr>
<tr>
<td>192.168.1.1</td>
<td>10.10.30.232 (IPv4)</td>
</tr>
<tr>
<td>FAA0::1</td>
<td>FAA0::0 (IPv6)</td>
</tr>
<tr>
<td>himalaya,gangotri</td>
<td>himalaya and gangotri</td>
</tr>
<tr>
<td>1@himalaya,1@gangotri</td>
<td>himalaya and gangotri (with a maximum participant ID of 1 on each host)</td>
</tr>
<tr>
<td>FAA0::0#localhost</td>
<td>FAA0::0#localhost (could be a UDPv4 transport plug-in registered at network address of FAA0::0) (IPv6)</td>
</tr>
<tr>
<td>udpv4://himalaya</td>
<td>himalaya accessed using the &quot;udpv4&quot; transport plug-in (IPv4)</td>
</tr>
<tr>
<td>udpv4://FAA0::0#localhost</td>
<td>localhost using the &quot;udpv4&quot; transport plug-in registered at network address FAA0::0</td>
</tr>
<tr>
<td>0/0/R</td>
<td>0/0/R (StarFabric)</td>
</tr>
</tbody>
</table>
Table 15.1 NDDS_DISCOVERY_PEERS Environment Variable Examples

<table>
<thead>
<tr>
<th>NDDS_DISCOVERY_PEERS</th>
<th>Description of Host(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>starfabric://0/0/R</td>
<td>0/0/R (StarFabric) using the &quot;starfabric&quot; (StarFabric) transport plug-ins</td>
</tr>
<tr>
<td>starfabric://#0/0/R</td>
<td>0/0/R (StarFabric) using the &quot;starfabric&quot; (StarFabric) transport plug-ins registered at network address FAA0::0</td>
</tr>
<tr>
<td>starfabric://FBB0::0#0/0/R</td>
<td>all unicast addresses accessed via the &quot;starfabric&quot; (StarFabric) transport plug-ins</td>
</tr>
<tr>
<td>shmem://FCC0::0</td>
<td>all unicast addresses accessed via the &quot;shmem&quot; (shared memory) transport plug-ins registered at network address FCC0::0</td>
</tr>
</tbody>
</table>

15.2.3 NDDS_DISCOVERY_PEERS File Format

You can set the default value for the initial peers list in a file named NDDS_DISCOVERY_PEERS. The file must be in your application’s current working directory.

The file is optional. If it is found, it supersedes the values in any environment variable of the same name.

Entries in the file must contain a sequence of peer descriptors separated by whitespace or the comma (',') character. The file may also contain comments starting with a semicolon (';') character until the end of the line.

Example file contents:

```
;; NDDS_DISCOVERY_PEERS - Discovery Configuration File
;; Multicast builtin.udpv4://239.255.0.1 ; default discovery multicast addr

;; Unicast
localhost,192.168.1.1 ; A comma can be used a separator
FAA0::1 FAA0::0#localhost ; Whitespace can be used as a separator
1@himalaya ; Max participant ID of 1 on 'himalaya'
1@gangotri

;; UDpv4
udpv4://himalaya ; 'himalaya' via 'udpv4' transport plugin(s)
udpv4://FAA0::0#localhost ; 'localhost' via 'udpv4' transport plugin
; registered at network address FAA0::0

;; Shared Memory
shmem:// ; All 'shmem' transport plugin(s)
builtin.shmem:// ; The builtin builtin 'shmem' transport plugin
shmem://FCC0::0 ; Shared memory transport plugin registered
; at network address FCC0::0

;; StarFabric
0/0/R ; StarFabric node 0/0/R
starfabric://0/0/R ; 0/0/R accessed via 'starfabric' transport plugin(s)
starfabric://FBB0::0#0/0/R ; StarFabric transport plugin registered
; at network address FBB0::0
starfabric:// ; All 'starfabric' transport plugin(s)
```
15.3 Discovery Implementation

Note: this section contains advanced material not required by most users.

Discovery is implemented using built-in DataWriters and DataReaders. These are the same class of entities your application uses to send/receive data. That is, they are also of type DDSDataWriter/DDSDataReader. For each DomainParticipant, three built-in DataWriters and three built-in DataReaders are automatically created for discovery purposes. Figure 15.2: Built-in Writers and Readers for Discovery below shows how these objects are used. (For more on built-in DataReaders and DataWriters, see Built-In Topics (Chapter 18 on page 831)).

Figure 15.2: Built-in Writers and Readers for Discovery

For each DomainParticipant, there are six objects automatically created for discovery purposes. The top two objects are used to send/receive participant DATA messages, which are used in the Participant Discovery phase to find remote DomainParticipants. This phase uses best-effort communications. Once the participants are aware of each other, they move on to the Endpoint Discovery Phase to learn about each other’s DataWriters and DataReaders. This phase uses reliable communications.

The implementation is split into two separate protocols:

Simple Participant Discovery Protocol (SPDP)
+ Simple Endpoint Discovery Protocol (SEDP)
= Simple Discovery Protocol (SDP)
15.3.1 Participant Discovery

When a DomainParticipant is created, a DataWriter and a DataReader are automatically created to exchange participant DATA messages in the network. These DataWriters and DataReaders are "special" because the DataWriter can send to a given list of destinations, regardless of whether there is a Connext DDS application at the destination, and the DataReader can receive data from any source, whether the source is previously known or not. In other words, these special readers and writers do not need to discover the remote entity and perform a match before they can communicate with each other.

When a DomainParticipant joins or leaves the network, it needs to notify its peer participants. The list of remote participants to use during discovery comes from the peer list described in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640. The remote participants are notified via participant DATA messages. In addition, if a participant’s QoS is modified in such a way that other participants need to know about the change (that is, changes to the 7.5.30 USER_DATA QosPolicy on page 465), a new participant DATA will be sent immediately.

Participant DATAs are also used to maintain a participant’s liveness status. These are sent at the rate set in the participant_liveliness_assert_period in the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644.

Let’s examine what happens when a new remote participant is discovered. If the new remote participant is in the local participant's peer list, the local participant will add that remote participant into its database. If the new remote participant is not in the local application's peer list, it may still be added, if the accept_unknown_peers field in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640 is set to TRUE.

Once a remote participant has been added to the Connext DDS database, Connext DDS keeps track of that remote participant’s participant_liveliness_lease_duration. If a participant DATA for that participant (identified by the GUID) is not received at least once within the participant_liveliness_lease_duration, the remote participant is considered stale, and the remote participant, together with all its entities, will be removed from the database of the local participant.

To keep from being purged by other participants, each participant needs to periodically send a participant DATA to refresh its liveness. The rate at which the participant DATA is sent is controlled by the participant_liveliness_assert_period in the participant’s 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644. This exchange, which keeps Participant A from appearing ‘stale,’ is illustrated in Figure 15.3: Periodic ‘participant DATAs’ on the next page. Figure 15.4: Ungraceful Termination of a Participant on page 776 shows what happens when Participant A terminates ungracefully and therefore needs to be seen as ‘stale.’
15.3.1 Participant Discovery

Figure 15.3: Periodic 'participant DATAs'

The DomainParticipant on Node A sends a 'participant DATA' to Node B, which is in Node A's peers list. This occurs regardless of whether or not there is a Connext DDS application on Node B.

The green short dashed lines are periodic participant DATAs. The time between these messages is controlled by the participant_liveliness_assert_period in the DiscoveryConfig QosPolicy.

In addition to the periodic participant DATAs, 'initial repeat messages' (shown in blue, with longer dashes) are sent from A to B. These messages are sent at a random time between min_initial_participant_announcement_period and max_initial_participant_announcement_period (in A's DiscoveryConfig QosPolicy). The number of these initial repeat messages is set in initial_participant_announcements.
15.3.1 Participant Discovery

Figure 15.4: Ungraceful Termination of a Participant

Participant A is removed from participant B’s database if it is not refreshed within the liveliness lease duration. Dashed lines are periodic participant DATA messages.

(Periodic resends of ‘participant B DATA’ from B to A are omitted from this diagram for simplicity. Initial repeat messages from A to B are also omitted from this diagram—these messages are sent at a random time between min_initial_participant_announcement_period and max_initial_participant_announcement_period, see Figure 15.3: Periodic ‘participant DATAs’ on the previous page.)
15.3.1.1 Refresh Mechanism

To ensure that a late-joining participant does not need to wait until the next refresh of the remote participant \textit{DATA} to discover the remote participant, there is a resend mechanism. If the received participant \textit{DATA} is from a never-before-seen remote participant, and it is in the local participant's peers list, the application will resend its own participant \textit{DATA} to all its peers. This resend can potentially be done multiple times, with a random sleep time in between. Figure 15.5: Resending ‘participant DATA’ to a Late-Joiner on the next page illustrates this scenario.

The number of retries and the random amount of sleep between them are controlled by each participant’s 9.5.3 DISCOVERY\_CONFIG QosPolicy (DDS Extension) on page 644 (see Figure 15.5: Resending ‘participant DATA’ to a Late-Joiner on the next page).
Participant A has Participant B in its peers list. Participant B does not have Participant A in its peers list, but [DiscoveryQosPolicy.accept_unknown_peers] is set to DDS_BOOLEAN_TRUE. Participant A joins the system after B has sent its initial announcement. After B discovers A, it waits for time $\Delta$, then resends its participant DATA.
15.3.1 Participant Discovery

(Initial repeat messages are omitted from this diagram for simplicity, see Figure 15.3: Periodic ‘participant DATAs’ on page 775.)

Figure 15.6: Participant Discovery Summary below provides a summary of the messages sent during the participant discovery phase.

Figure 15.6: Participant Discovery Summary

Participants A and B both have each other in their peers lists. Participant A is created first.

15.3.1.2 Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT

To maintain the liveliness of DataWriters that have a 7.5.15 LIVELINESS QosPolicy on page 427 kind field set to AUTOMATIC or MANUAL_BY_PARTICIPANT, Connext DDS uses a built-in DataWriter and DataReader pair, referred to as the inter-participant reader and inter-participant writer.

If the DomainParticipant has any DataWriters with Liveliness QosPolicy kind set to AUTOMATIC, the inter-participant writer will reliably broadcast an AUTOMATIC Liveliness message at a period equal to X, where X is the shortest assertion period of these DataWriters. (The assertion period for a DataWriter is
calculated as \texttt{lease\_duration / assertions\_per\_lease\_duration}, which are fields in the 7.5.15 LIVELINESS QosPolicy on page 427.) Figure 15.7: DataWriter with AUTOMATIC Liveliness below illustrates this scenario.

Figure 15.7: DataWriter with AUTOMATIC Liveliness

A Liveliness message is sent automatically when a DataWriter with AUTOMATIC Liveliness kind is created, and then periodically.

If the \texttt{DomainParticipant} has any \texttt{DataWriters} with Liveliness QosPolicy \texttt{kind} set to \texttt{MANUAL\_BY\_PARTICIPANT}, Connext DDS will periodically check to see if any of them have called \texttt{write()}, \texttt{assert\_liveliness()}, \texttt{dispose()} or \texttt{unregister()}. The rate of this check is every \(X/2\) seconds, where \(X\) is the smallest \texttt{lease\_duration} among all the \texttt{DomainParticipant}'s \texttt{MANUAL\_BY\_PARTICIPANT} \texttt{DataWriters}. (The \texttt{lease\_duration} is a field in the 7.5.15 LIVELINESS QosPolicy on page 427.) If any
of the MANUAL_BY_PARTICIPANT DataWriters have called any of those operations, the inter-participant writer will reliably broadcast a MANUAL Liveliness message.

If a DomainParticipant's assert_liveliness() operation is called, and that DomainParticipant has any MANUAL_BY_PARTICIPANT DataWriters, the inter-participant writer will reliably broadcast a MANUAL Liveliness message within the above-defined X/2 time period. These MANUAL Liveliness messages are used to update the liveliness of all the DomainParticipant's MANUAL_BY_PARTICIPANT DataWriters. As described in 7.5.15.1 Timing Considerations for MANUAL_BY_PARTICIPANT on page 430, Connext DDS applications should make sure to assert liveliness at a period that is shorter than half of the minimum lease duration; otherwise, liveliness might be lost for the DataWriter. Figure 15.8: DataWriter with MANUAL_BY_PARTICIPANT Liveliness on the next page shows an example sequence.
Once a MANUAILBYPARTICIPANT DataWriter is created, subsequent calls to assert_liveliness, write, dispose, or unregister_instance will trigger Liveliness messages, which update the liveliness status of all the participant’s DataWriters.

The inter-participant reader receives data from remote inter-participant writers and asserts the liveliness of remote DomainParticipants endpoints accordingly.

If the DomainParticipant has no DataWriters with 7.5.15 LIVELINESS QosPolicy on page 427 kind set to AUTOMATIC or MANUALBYPARTICIPANT, then no Liveliness messages are ever sent from the inter-participant writer.
15.3.2 Endpoint Discovery

As we saw in Figure 15.2: Built-in Writers and Readers for Discovery on page 773, reliable DataReaders and DataWriters are automatically created to exchange publication/subscription information for each DomainParticipant. We will refer to these as ‘discovery endpoint readers and writers.’ However, nothing is sent through the network using these entities until they have been ‘matched’ with their remote counterparts. This ‘matching’ is triggered by the Participant Discovery phase. The goal of the Endpoint Discovery phase is to add the remote endpoint to the local database, so that user-created endpoints (your application’s DataWriters/DataReaders) can communicate with each other.

When a new remote DomainParticipant is discovered and added to a participant’s database, Connext DDS assumes that the remote DomainParticipant is implemented in the same way and therefore is creating the appropriate counterpart entities. Therefore, Connext DDS will automatically add two remote discovery endpoint readers and two remote discovery endpoint writers for that remote DomainParticipant into the local database. Once that is done, there is now a match with the local discovery endpoint writers and readers, and publication DATAs and subscription DATAs can then be sent between the discovery endpoint readers/writers of the two DomainParticipants.

When you create a DataWriter/DataReader for your user data, a publication/subscription DATA describing the newly created object is sent from the local discovery endpoint writer to the remote discovery endpoint readers of the remote DomainParticipants that are currently in the local database.

If your application changes any of the following QosPolicies for a local user-data DataWriter/DataReader, a modified subscription/publication DATA is sent to propagate the QoS change to other DomainParticipants:

- 5.2.1 TOPIC_DATA QosPolicy on page 224
- 7.4.4 GROUP_DATA QosPolicy on page 353
- 7.5.30 USER_DATA QosPolicy on page 465
- 7.5.18 OWNERSHIP_STRENGTH QosPolicy on page 438
- 7.4.5 PARTITION QosPolicy on page 356
- 8.6.4 TIME_BASED_FILTER QosPolicy on page 588
- 7.5.14 LIFESPAN QoS Policy on page 425

What the above QosPolicies have in common is that they are all changeable and part of the built-in data (see Built-In Topics (Chapter 18 on page 831)).

Similarly, if the application deletes any user-data writers/readers, the discovery endpoint writer/readers send delete publication/subscription DATAs. In addition to sending publication/subscription DATAs, the discovery endpoint writer will check periodically to see if the remote discovery endpoint reader is up-to-date. (The rate for this check is the publication_writer.heartbeat_period or subscription_writer.heartbeat_period in the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644. If the discovery
endpoint writer has not been acknowledged by the remote discovery endpoint reader regarding receipt of the latest DATA, the discovery endpoint writer will send a special Heartbeat (HB) message with the Final bit set to 0 (F=0) to request acknowledgement from the remote discovery endpoint reader, as seen in Figure 15.9: Endpoint Discovery Summary below.

**Figure 15.9: Endpoint Discovery Summary**

---

1. DataWriter C created
2. Periodic HB not sent since A knows that B is up-to-date
3. Modify DataWriter C's UserData QoS
4. DataWriter C deleted

---

Node A

1. Publication C DATA
2. Periodic HB (F=0)
3. ACKNACK
4. Modified Publication C DATA
5. Periodic HB (F=0)
6. ACKNACK
7. Delete Publication C DATA
8. Periodic HB (F=0)
9. ACKNACK

---

Node B

1. Remote DataWriter C discovered, added to database
2. DataWriter C's QoS modified, Database record modified
3. DataWriter C removed from database

---

0 = A's DDS_DomainParticipantQos.discovery_config.publication_writer.heartbeat_period

1 = wait random time between B's DDS_DomainParticipantQos.discovery_config.publication_reader.min_heartbeat_response_delay and [... max_heartbeat_response_delay]
Assume participants A and B have been discovered on both sides. A’s DiscoveryConfigQosPolicy.publication_writer.heartbeats_per_max_samples = 0, so no HB is piggybacked with the publication DATA. A HB with F=0 is a request for an ACK/NACK. The periodic and initial repeat participant DATAs are omitted from the diagram.

Discovery endpoint writers and readers have their 7.5.12 HISTORY QosPolicy on page 420 set to KEEP_LAST, and their 7.5.9 DURABILITY QosPolicy on page 411 set to TRANSIENT_LOCAL. Therefore, even if the remote DomainParticipant has not yet been discovered at the time the local user’s DataWriter/DataReader is created, the remote DomainParticipant will still be informed about the previously created DataWriter/DataReader. This is achieved by the HB and ACK/NACK that are immediately sent by the built-in endpoint writer and built-in endpoint reader respectively when a new remote participant is discovered. Figure 15.10: DataWriter Discovered by Late-Joiner, Triggered by HB below and Figure 15.11: DataWriter Discovered by Late-Joiner, Triggered by ACKNACK on the next page illustrate this sequence for HB and ACK/NACK triggers, respectively.

**Figure 15.10: DataWriter Discovered by Late-Joiner, Triggered by HB**

**Figure 15.11: DataWriter Discovered by Late-Joiner, Triggered by ACKNACK**

**Writer C is created on Participant A before Participant A discovers Participant B. Assuming DiscoveryConfigQosPolicy.publication_writer.heartbeats_per_max_samples = 0, no HB is piggybacked with the publication DATA. Participant B has A in its peer list, but not vice versa. Accept_unknown_locators is true. On A, in response to**
receiving the new participant B DATA message, a participant A DATA message is sent to B. The discovery endpoint reader on A will also send an ACK/NACK to the discovery endpoint writer on B. (Initial repeat participant messages and periodic participant messages are omitted from this diagram for simplicity, see Figure 15.3: Periodic 'participant DATAs' on page 775 in 15.3.1 Participant Discovery on page 774.)

Figure 15.11: DataWriter Discovered by Late-Joiner, Triggered by ACKNACK

Endpoint discovery latency is determined by the following members of the DomainParticipant’s 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644:

- publication_writer
- subscription_writer
When a remote entity record is added, removed, or changed in the database, matching is performed with all the local entities. Only after there is a successful match on both ends can an application’s user-created *DataReaders* and *DataWriters* communicate with each other.

For more information about reliable communication, see *Reliable Communications* (Chapter 11 on page 688).

### 15.3.3 Discovery Traffic Summary

This diagram shows both phases of the discovery process. Participant A is created first, followed by Participant B. Each has the other in its peers list. After they have discovered each other, a DataWriter is created on Participant A. Periodic participant DATAs, HBs and ACK/NACKs are omitted from this diagram.

Node A

1. Participant A created

2. Participant B created

Node B

1. First repeat of participant A DATA

2. First repeat of participant B DATA

- **publication_reader**
- **subscription_reader**

For more information about reliable communication, see *Reliable Communications* (Chapter 11 on page 688).
15.3.4 Discovery-Related QoS

Each DomainParticipant needs to be uniquely identified in the DDS domain and specify which other DomainParticipants it is interested in communicating with. The 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670 uniquely identifies a DomainParticipant in the DDS domain. The 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640 specified the peer participants it is interested in communicating with.

There is a trade-off between the amount of traffic on the network for the purposes of discovery and the delay in reaching steady state when the DomainParticipant is first created.

For example, if the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640’s participant_liveliness_assert_period and participant_liveliness_lease_duration fields are set to small values, the discovery of stale remote DomainParticipants will occur faster, but more discovery traffic will be sent over the network. Setting the participant’s heartbeat_period to a small value can cause late-joining DomainParticipants to discover remote user-data DataWriters and DataReaders at a faster rate, but Connext DDS might send HBs to other nodes more often. This timing can be controlled by the following DomainParticipant QosPolicies:

- 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640 — specifies how other DomainParticipants in the network can communicate with this DomainParticipant, and which other DomainParticipants in the network this DomainParticipant is interested in communicating with. See also: 15.5 Ports Used for Discovery on page 791.

- 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644 — specifies the QoS of the discovery readers and writers (parameters that control the HB and ACK rates of discovery end-point readers/writers, and periodic refreshing of participant DATA from discovery participant readers/writers). It also allow you to configure asynchronous writers in order to send data with a larger size than the transport message size.

- 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654 — specifies the number of local and remote entities expected in the system.

- 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670 — specifies the rtps_app_id and rtps_host_id that uniquely identify the participant in the DDS domain.

The other important parameter is the domain ID: DomainParticipants can only discover each other if they belong to the same DDS domain. The domain ID is a parameter passed to the create_participant() operation (see 9.3.1 Creating a DomainParticipant on page 615).

---

 heartbeat_period is part of the DDS_RtpsReliableWriterProtocol_t structure used in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640’s publication_writer and subscription_writer fields.
15.4 Debugging Discovery

To understand the flow of messages during discovery, you can increase the verbosity of the messages logged by Connext DDS so that you will see whenever a new entity is discovered, and whenever there is a match between a local entity and a remote entity.

This can be achieved with the logging API:

```cpp
NDDSConfigLogger::get_instance()->set_verbosity_by_category (NDDS_CONFIG_LOG_CATEGORY_ENTITIES,
NDDS_CONFIG_LOG_VERBOSITY_STATUS_REMOTE);
```

Using the scenario in the summary diagram in 15.3.3 Discovery Traffic Summary on page 787, these are the messages as seen on DomainParticipant A:

```plaintext
[D0049|ENABLE]DISCPluginManager_onAfterLocalParticipantEnabled:announcing new local participant: 0XA0A01A1,0X5522,0X1,0X1C1
[D0049|ENABLE]DISCPluginManager_onAfterLocalParticipantEnabled:at {46c614d9,0C43B2DC}

(The above messages mean: First participant A DATA sent out when participant A is enabled.)

DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:discovered new participant:
host=0x0A0A01A1, app=0x0000552B, instance=0x00000001
DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:at {46c614dd,8FA13C1F}
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin discovered/updated remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at {46c614dd,8FACE677}
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin accepted new remote participant:
0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at {46c614dd,8FACE677}

(The above messages mean: Received participant B DATA.)

DISCSimpleParticipantDiscoveryPlugin_remoteParticipantDiscovered:re-announcing participant self: 0XA0A01A1,0X5522,0X1,0X1C1
DISCSimpleParticipantDiscoveryPlugin_remoteParticipantDiscovered:at {46c614dd,8FC02AF7}

(The above messages mean: Resending participant A DATA to the newly discovered remote participant.)

PRESPsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X200C2, local 0x000200C7 in reliable reader service
PRESPsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,0X1,0X200C7, local 0x000200C2 in reliable writer service
PRESPsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,0X1,0X4C7, local 0x000004C2 in reliable writer service
PRESPsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,0X1,0X3C7, local 0x000003C2 in reliable writer service
PRESPsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X4C2, local 0x000004C7 in reliable reader service
PRESPsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X3C2, local 0x000003C7 in reliable reader service
PRESPsService_linkToLocalReader:assert remote 0XA0A01A1,0X552B,0X1,0X100C2, local 0x000100C7 in best effort reader service

(The above messages mean: Automatic matching of the discovery readers and writers. A built-in remote endpoint's object ID always ends with Cx.)
DISCSimpleParticipantDiscoveryPluginReaderListener.onDataAvailable: discovered modified participant: host=0x0A0A01A1, app=0x0000552B, instance=0x00000001
DISCParticipantDiscoveryPlugin_assertRemoteParticipant: plugin discovered/updated remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at {46c614dd,904D876C}

(The above messages mean: Received participant B DATA.)

DISCPluginManager.onAfterLocalEndpointEnabled: announcing new local publication: 0XA0A01A1,0X5522,0X1,0X80000003
DISCPluginManager.onAfterLocalEndpointEnabled:at {46c614d9,1013B9F0}
DISCSimpleEndpointDiscoveryPluginPDFListener.onAfterLocalWriterEnabled: announcing new publication: 0XA0A01A1,0X5522,0X1,0X80000003
DISCSimpleEndpointDiscoveryPluginPDFListener.onAfterLocalWriterEnabled:at {46c614d9,101615EB}

(The above messages mean: Publication C DATA has been sent.)

DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAvailable: discovered subscription: 0XA0A01A1,0X5522,0X1,0X80000004
DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAvailable:at {46c614dd,94FAEFEF}
DISCEndpointDiscoveryPlugin_assertRemoteEndpoint: plugin discovered/updated remote endpoint: 0XA0A01A1,0X5522,0X1,0X80000004
DISCEndpointDiscoveryPlugin_assertRemoteEndpoint:at {46c614dd,950203DF}

(The above messages mean: Receiving subscription D DATA from Node B.)

PRESpsService_linkToLocalWriter: assert remote 0XA0A01A1,0X5522,0X1,0X80000004, local 0x80000003 in best effort writer service

(The above message means: User-created DataWriter C and DataReader D are matched.)

[D0049|DELETE_CONTAINED] DISCPluginManager.onAfterLocalEndpointDeleted: announcing disposed local publication: 0XA0A01A1,0X5522,0X1,0X80000003
[D0049|DELETE_CONTAINED] DISCPluginManager.onAfterLocalEndpointDeleted:at {46c61501,28A11663}
[D0049|DELETE_CONTAINED] DISCPluginManager.onAfterLocalEndpointDeleted: announcing disposed publication: 0XA0A01A1,0X5522,0X1,0X80000003
[D0049|DELETE_CONTAINED] DISCPluginManager.onAfterLocalEndpointDeleted:at {46c61501,28840E15}

(The above messages mean: Publication C DATA(delete) has been sent.)

DISCPluginManager.onBeforeLocalParticipantDeleted: announcing before disposed local participant: 0XA0A01A1,0X5522,0X1,0X1C1
DISCPluginManager.onBeforeLocalParticipantDeleted:at {46c61501,28A11663}

(The above messages mean: Participant A DATA(delete) has been sent.)
15.5 Ports Used for Discovery

There are two kinds of traffic in a Connext DDS application: discovery (meta) traffic, and user traffic. Meta-traffic is for data (declarations) that is sent between the automatically-created discovery writers and readers; user traffic is for data that is sent between user-created DataWriters and DataReaders. To keep the two kinds of traffic separate, Connext DDS uses different ports, as described below.

**Note:** The ports described in this section are used for incoming data. Connext DDS uses ephemeral ports for outbound data.

Connext DDS uses the RTPS wire protocol. The discovery protocols defined by RTPS rely on well-known ports to initiate discovery. These well-known ports define the multicast and unicast ports on which a Participant will listen for meta-traffic from other Participants. The meta-traffic contains the information required by Connext DDS to establish the presence of remote Entities in the network.

The well-known incoming ports are defined by RTPS in terms of port mapping expressions with several tunable parameters. This allows you to customize what network ports are used for receiving data by Connext DDS. These parameters are shown in Table 15.2 WireProtocol QosPolicy’s rtps_well_known_ports (DDS_RtpsWellKnownPorts_t). (For defaults and valid ranges, please see the API Reference HTML documentation.)
In order for all Participants in a system to correctly discover each other, it is important that they all use the same port mapping expressions.

In addition to the parameters listed in Table 15.2 WireProtocol QosPolicy’s rtps_well_known_ports (DDS_RtpsWellKnownPorts_t), the port formulas described below depend on:

- The domain ID specified when the DomainParticipant is created (see 9.3.1 Creating a DomainParticipant on page 615). The domain ID ensures no port conflicts exist between Participants belonging to different domains. This also means that discovery traffic in one DDS domain is not visible to DomainParticipants in other DDS domains.

- The participant_id is a field in the 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670, see 9.5.9.1 Choosing Participant IDs on page 671. The participant_id ensures that unique unicast port numbers are assigned to DomainParticipants belonging to the same DDS domain on a given host.

**Backwards Compatibility:** Connext DDS supports the standard DDS Interoperability Wire Protocol based on the Real-time Publish-Subscribe (RTPS) protocol. This protocol is not compatible with the one used by earlier releases (4.2c or lower). Therefore, applications built with 4.2d or higher will not interoperate with applications built with 4.2c or lower. The default port mapping from domainID and participant index has also been changed according to the new interoperability specification. The message types and formats used by RTPS have also changed.

**Port Aliasing:** When modifying the port mapping parameters, avoid port aliasing. This would result in undefined discovery behavior. The chosen parameter values will also determine the maximum possible number of DDS domains in the system and the maximum number of participants per DDS domain.
Additionally, any resulting mapped port number must be within the range imposed by the underlying transport. For example, for UDPv4, this range typically equals [1024 - 65535].

### 15.5.1 Inbound Ports for Meta-Traffic

The Wire Protocol QosPolicy’s `rtps_well_known_ports.metatraffic_unicast_port` determines the port used for receiving meta-traffic using unicast:

\[
\text{metatraffic\_unicast\_port} = \text{port\_base} + \\
\quad (\text{domain\_id\_gain} \times \text{Domain ID}) + \\
\quad (\text{participant\_id\_gain} \times \text{participant\_id}) + \\
\quad \text{builtin\_unicast\_port\_offset}
\]

Similarly, `rtps_well_known_ports.metatraffic_multicast_port` determines the port used for receiving meta-traffic using multicast. The corresponding multicast group addresses are specified via `multicast\_receive\_addresses` (see 9.5.2.4 Configuring Multicast Receive Addresses on page 642).

\[
\text{metatraffic\_multicast\_port} = \text{port\_base} + \\
\quad (\text{domain\_id\_gain} \times \text{Domain ID}) + \\
\quad \text{builtin\_multicast\_port\_offset}
\]

**Note:** Multicast is only used for meta-traffic if a multicast address is specified in the NDDS_DISCOVERY_PEERS environment variable or file or if the `multicast\_receive\_addresses` field of the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644 is set.

### 15.5.2 Inbound Ports for User Traffic

RTPS also defines the default multicast and unicast ports on which `DataReaders` and `DataWriters` receive user traffic. These default ports can be overridden using the `DataReader’s 8.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension)` on page 591 and `7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension)` on page 460, or the `DataWriter’s 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension)` on page 460.

The WireProtocol QosPolicy’s `rtps_well_known_ports.usertraffic_unicast_port` determines the port used for receiving user data using unicast:

\[
\text{usertraffic\_unicast\_port} = \\
\quad \text{port\_base} + \\
\quad (\text{domain\_id\_gain} \times \text{Domain ID}) + \\
\quad (\text{participant\_id\_gain} \times \text{participant\_id}) + \\
\quad \text{user\_unicast\_port\_offset}
\]

Similarly, `rtps_well_known_ports.usertraffic_multicast_port` determines the port used for receiving user data using multicast. The corresponding multicast group addresses can be configured using the `7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension)` on page 460.
15.5.3 Automatic Selection of participant_id and Port Reservation

The 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670 field determines what type of ports are reserved when the DomainParticipant is enabled. See 9.5.9.1 Choosing Participant IDs on page 671.

15.5.4 Tuning domain_id_gain and participant_id_gain

The domain_id_gain is used as a multiplier of the domain ID. Together with participant_id_gain (15.5.4 Tuning domain_id_gain and participant_id_gain above), these values determine the highest domain ID and participant_id allowed on this network.

In general, there are two ways to set up the domain_id_gain and participant_id_gain parameters.

- If domain_id_gain > participant_id_gain, it results in a port mapping layout where all DomainParticipants in a DDS domain occupy a consecutive range of domain_id_gain ports. Precisely, all ports occupied by the DDS domain fall within:

\[
\text{port_base + (domain_id_gain * Domain ID)}
\]

and:

\[
\text{port_base + (domain_id_gain * (Domain ID + 1)) - 1}
\]

In this case, the highest domain ID is limited only by the underlying transport's maximum port. The highest participant_id, however, must satisfy:

\[
\text{maxParticipant_id < (domain_id_gain / participant_id_gain)}
\]

- Or if domain_id_gain <= participant_id_gain, it results in a port mapping layout where a given DDS domain's DomainParticipant instances occupy ports spanned across the entire valid port range allowed by the underlying transport. For instance, it results in the following potential mapping:

<table>
<thead>
<tr>
<th>Mapped Port</th>
<th>Domain ID</th>
<th>Participant ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>higher port number</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lower port number</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
In this case, the highest `participant_id` is limited only by the underlying transport's maximum port. The highest `domain_id`, however, must satisfy:

\[
\text{max\_domain\_id} < (\text{participant\_id\_gain} / \text{domain\_id\_gain})
\]

The `domain_id_gain` also determines the range of the port-specific offsets:

\[
\text{domain\_id\_gain} > \abs{\text{builtin\_multicast\_port\_offset} - \text{user\_multicast\_port\_offset}}
\]

and

\[
\text{domain\_id\_gain} > \abs{\text{builtin\_unicast\_port\_offset} - \text{user\_unicast\_port\_offset}}
\]

Violating this may result in port aliasing and undefined discovery behavior.

The `participant_id_gain` also determines the range of `builtin\_unicast\_port\_offset` and `user\_unicast\_port\_offset`:

\[
\text{participant\_id\_gain} > \abs{\text{builtin\_unicast\_port\_offset} - \text{user\_unicast\_port\_offset}}
\]

In all cases, the resulting ports must be within the range imposed by the underlying transport.
Chapter 16 Transport Plugins

Connext DDS has a pluggable-transport architecture. The core of Connext DDS is transport agnostic—it does not make any assumptions about the actual transports used to send and receive messages. Instead, Connext DDS uses an abstract "transport API" to interact with the transport plugins that implement that API. A transport plugin implements the abstract transport API, and performs the actual work of sending and receiving messages over a physical transport.

There are essentially three categories of transport plugins:

- **Builtin Transport Plugins** Connext DDS comes with a set of commonly used transport plugins. These ‘builtin’ plugins include UDPv4, UDPv6, and shared memory. So that Connext DDS applications can work out-of-the-box, some of these are enabled by default (see 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665).

- **Extension Transport Plugins** RTI offers extension transports, including RTI Secure WAN Transport (see Part 6: RTI Secure WAN Transport on page 1109 and RTI TCP Transport (see Part 7: RTI TCP Transport on page 1142).

- **Custom-developed Transport Plugins** RTI supports the use of custom transport plugins. This is a powerful capability that distinguishes Connext DDS from competing middleware approaches. If you are interested in developing a custom transport plugin for Connext DDS, please contact your local RTI representative or email sales@rti.com.

16.1 Builtin Transport Plugins

There are two ways in which the builtin transport plugins may be registered:

- **Default builtin Transport Instances**: Builtin transports that are turned "on" in the 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665 are implicitly registered when (a) the DomainParticipant is enabled, (b) the first DataWriter/DataReader is created, or (c) you look up a builtin DataReader (by calling lookup_datareader() on a Subscriber), whichever happens first. The builtin transport plugins have default properties. If
you want to change these properties, do so before the transports are registered.

- **Other Transport Instances**: There are two ways to install non-default built-in transport instances:
  - Transport plugins may be explicitly registered by first creating an instance of the transport plugin (by calling `NDDS_Transport_UDPv4_new()`, `NDDS_Transport_UDPv6_new()` or `NDDS_Transport_Shimem_new()`), see 16.4 Explicitly Creating Builtin Transport Plugin Instances on the next page, then calling `register_transport()` (16.7 Installing Additional Builtin Transport Plugins with `register_transport()` on page 821). (For example, suppose you want an extra instance of a transport.) (Not available for the Java or .NET API.)
  - Additional built-in transport instances can also be installed through the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.

To configure the properties of the built-in transports:

- Set properties by calling `set_builtin_transport_property()` (see 16.5 Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties() on page 799)

or

- Specify predefined property strings in the *DomainParticipant*'s `PropertyQosPolicy`, as described in 16.6 Setting Builtin Transport Properties with the `PropertyQosPolicy` on page 801.

For other built-in transport instances:

- If the built-in transport plugin is created with `NDDS_Transport_UDPv4_new()`, `NDDS_Transport_UDPv6_new()` or `NDDS_Transport_Shimem_new()`, properties can be specified during creation time. See 16.4 Explicitly Creating Builtin Transport Plugin Instances on the next page.
- If the additional built-in transport instances are installed through the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439, the properties of the built-in transport plugins can also be specified through that same QosPolicy.

### 16.2 Extension Transport Plugins

If you want to change the properties for an extension transport plugin, do so before the plugin is registered. Any transport property changes made after the plugin is registered will have no effect.

There are two ways to install an extension transport plugin:

---

1. Any transport property changes made after the plugin is registered will have no effect.
• **Implicit Registration**: Transports can be installed through the predefined strings in the DomainParticipant’s PropertyQosPolicy. Once the transport’s properties are specified in the PropertyQosPolicy, the transport will be implicitly registered when (a) the DomainParticipant is enabled, (b) the first DataWriter/DataReader is created, or (c) you look up a builtin DataReader (by calling `lookup_datareader()` on a Subscriber), whichever happens first.

QoS policies can also be configured from XML resources (files, strings)—with this approach, you can change the QoS without recompiling the application. The QoS settings are automatically loaded by the DomainParticipantFactory when the first DomainParticipant is created. For more information, see Configuring QoS with XML (Chapter 19 on page 848).

• **Explicit Registration**: Transports may be explicitly registered by first creating an instance of the transport plugin (see 16.4 Explicitly Creating Builtin Transport Plugin Instances below) and then calling `register_transport()` (see 16.7 Installing Additional Builtin Transport Plugins with `register_transport()` on page 821).

### 16.3 The NDDSTransportSupport Class

The `register_transport()` and `set_builtin_transport_property()` operations are part of the NDDSTransportSupport class, which includes the operations listed in Table 16.1 Transport Support Operations.

#### Table 16.1 Transport Support Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_transport_plugin</code></td>
<td>Retrieves a previously registered transport plugin.</td>
<td>16.7 Installing Additional Builtin Transport Plugins with <code>register_transport()</code> on page 821</td>
</tr>
<tr>
<td><code>register_transport</code></td>
<td>Registers a transport plugin for use with a DomainParticipant.</td>
<td></td>
</tr>
<tr>
<td><code>get_builtin_transport_property</code></td>
<td>Gets the properties used to create a builtin transport plugin.</td>
<td>16.5 Setting Builtin Transport Properties of Default Transport Instance–<code>get/set_builtin_transport_properties()</code> on the next page</td>
</tr>
<tr>
<td><code>set_builtin_transport_property</code></td>
<td>Sets the properties used to create a builtin transport plugin.</td>
<td></td>
</tr>
<tr>
<td><code>add_send_route</code></td>
<td>Adds a route for outgoing messages.</td>
<td>16.9.1 Adding a Send Route on page 824</td>
</tr>
<tr>
<td><code>add_receive_route</code></td>
<td>Adds a route for incoming messages.</td>
<td>16.9.2 Adding a Receive Route on page 825</td>
</tr>
<tr>
<td><code>lookup_transport</code></td>
<td>Looks up a transport plugin within a DomainParticipant.</td>
<td>16.9.3 Looking Up a Transport Plugin on page 826</td>
</tr>
</tbody>
</table>

### 16.4 Explicitly Creating Builtin Transport Plugin Instances

The builtin transports (UDPv4, UDPv6, and Shared Memory) are implicitly created by default (if they are enabled via the 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665). Therefore,
you only need to explicitly create a new instance if you want an extra instance (suppose you want two UDPv4 transports, one with special settings).

Transport plugins may be explicitly registered by first creating an instance of the transport plugin and then calling register_transport() (16.7 Installing Additional Builtin Transport Plugins with register_transport() on page 821). (For example, suppose you want an extra instance of a transport.) (Not available for the Java API.)

To create an instance of a builtin transport plugin, use one of the following functions:

```
NDDS_Transport_Plugin* NDDS_Transport_UDPv4_new (  
    const struct NDDS_Transport_UDPv4_Property_t * property_in)
NDDS_Transport_Plugin* NDDS_Transport_UDPv4_new (  
    const struct NDDS_Transport_UDPv4_Property_t * property_in)
NDDS_Transport_Plugin* NDDS_Transport_Shimem_new (  
    const struct NDDS_Transport_Shimem_Property_t * property_in)
```

Where:

```
property_in Desired behavior of this transport. May be NULL for default properties.
```

For details on using these functions, please see the API Reference HTML documentation.

Your application may create and register multiple instances of these transport plugins with Connext DDS. This may be done to partition the network interfaces across multiple DDS domains. However, note that the underlying transport, the operating system's IP layer, is still a "singleton." For example, if a unicast transport has already bound to a port, and another unicast transport tries to bind to the same port, the second attempt will fail.

### 16.5 Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties()

Perhaps you want to use one of the builtin transports, but need to modify the properties. (For default values, please see the API Reference HTML documentation.) Used together, the two operations below allow you to customize properties of the builtin transport when it is implicitly registered (see 16.1 Builtin Transport Plugins on page 796).

**Note:** Another way to change the properties is with the Property QosPolicy, see 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy on page 801. Changing properties with the Property QosPolicy will overwrite the properties set by calling set_builtin_transport_property().

```
DDS_ReturnCode_t  
NDDSTransportSupport::get_builtin_transport_property(  
    DDSDomainParticipant * participant_in,  
    DDS_TransportBuiltinKind builtin_transport_kind_in,  
    struct NDDS_Transport_Property_t  
    &builtin_transport_property_inout)
```

```
DDS_ReturnCode_t  
NDDSTransportSupport::set_builtin_transport_property(  
    DDSDomainParticipant * participant_in,  
    DDS_TransportBuiltinKind builtin_transport_kind_in,  
    struct NDDS_Transport_Property_t  
    &builtin_transport_property_inout)
```
16.5 Setting Built-in Transport Properties of Default Transport Instance—get/set_buildin_transport_

DDS.DomainParticipant * participant_in,
DDS_Transport.BuiltinKind builtin_transport_kind_in,
const struct NDDS_Transport_Property_t
&builtin_transport_property_in)

Where:

participant_in A valid non-NULL DomainParticipant that has not been enabled. If the DomainParticipant is already enabled when this operation is called, your transport property changes will not be reflected in the transport used by the DomainParticipant's DataWriters and DataReaders.

builtin_transport_kind_in The builtin transport kind for which to specify the properties.

builtin_transport_property_ (Used by the "get" operation only.) The storage area where the retrieved property will be output. The specific type required by the builtin_transport_kind_in must be used.

inout (Used by the "set" operation only.) The new transport property that will be used to create the builtin transport plugin. The specific type required by the builtin_transport_kind_in must be used.

In this example, we want to use the builtin UDPv4 transport, but with modified properties.

/* Before this point, create a disabled DomainParticipant */
struct NDDS_Transport_UDPv4_Property_t property =
NDDS_TRANSPORT_UDPV4_PROPERTY_DEFAULT;
if (NDDSTransportSupport::get_builtin_transport_property(
    participant, DDS.TRANSPORTBUILTIN_UDPv4,
    (struct NDDS_Transport_Property_t *)&property) !=
    DDS_RETCODE_OK) {
    printf("**Error: get builtin transport property\n");
}
/* Make your desired changes here */
/* For example, to increase the UDPv4 max msg size to 64K: */
property.parent.message_size_max = 65535;
property.recv_socket_buffer_size = 65535;
property.send_socket_buffer_size = 65535;
if (NDDSTransportSupport::set_builtin_transport_property(
    participant, DDS.TRANSPORTBUILTIN_UDPv4,
    (struct NDDS_Transport_Property_t *)&property)
    != DDS_RETCODE_OK) {
    printf("**Error: set builtin transport property\n");
}
/* Enable the participant to turn on communications with other participants in the DDS domain using the new properties for the automatically registered builtin transport plugins */
if (entity->enable() != DDS_RETCODE_OK) {
    printf("**Error: failed to enable entity\n");
}

Note: Built-in transport property changes will have no effect after the builtin transport has been registered. The builtin transports are implicitly registered when (a) the DomainParticipant is enabled, (b) the first DataWriter/DataReader is created, or (c) you lookup a builtin DataReader, whichever happens first.
16.6 Setting Builtin Transport Properties with the PropertyQosPolicy

The 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 allows you to set name/value pairs of data and attach them to an entity, such as a DomainParticipant.

To assign properties, use the add_property() operation:

```c
DDS_ReturnCode_t DDSPropertyQosPolicyHelper::add_property
(DDS_PropertyQosPolicy policy,
 const char * name,
 const char * value,
 DDS_Boolean propagate)
```

For more information on add_property() and the other operations in the DDSPropertyQosPolicyHelper class, please see Table 7.65 PropertyQoSPolicyHelper Operations, as well as the API Reference HTML documentation.

The ‘name’ part of the name/value pairs is a predefined string. The property names for the builtin transports are described in these tables:

- Table 16.2 Properties for the Builtin UDPv4 Transport
- Table 16.3 Properties for Builtin UDPv6 Transport
- Table 16.4 Properties for Builtin Shared-Memory Transport

See also:

- 16.6.1 Setting the Maximum Gather-Send Buffer Count for UDP Transports on page 819
- 16.6.2 Formatting Rules for IPv6 ‘Allow’ and ‘Deny’ Address Lists on page 820

**Note:**

Changing properties with the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 will overwrite any properties set by calling set_builtin_transport_property().

### Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv4.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.properties_bitmap</td>
<td>A bitmap that defines various properties of the transport to the Connext DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connext DDS tries to receive a message using the plugin. This is in support of a zero-copy interface. Default: 0</td>
</tr>
</tbody>
</table>
### Table 16.2 Properties for the builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv4.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
</table>
| parent.gather_send_buffer_count_max                          | Specifies the maximum number of buffers that Connext DDS can pass to the send() method of a transport plug-in.  
  The transport plugin send() API supports a gather-send concept, where the send() call can take several discontiguous buffers, assemble and send them in a single message. This enables Connext DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.  
  However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connext DDS from trying to gather too many buffers into a send call for the transport plugin.  
  Connext DDS requires all transport-plugin implementations to support a gather-send of at least a minimum number of buffers. This minimum number is NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.  
  See 16.6.1 Setting the Maximum Gather-Send Buffer Count for UDP Transports on page 819.  
  Default: 16 |
| parent.message_size_max                                     | The maximum size of a message in bytes that can be sent or received by the transport plug-in. Above this size, DDS-level fragmentation will occur. See 23.3 Large Data Fragmentation on page 970.  
  This value must be set before the transport plug-in is registered, so that Connext DDS can properly use the plug-in.  
  Default for Integrity platforms: 9216  
  Default for non-Integrity platforms: 65507 |
| parent.allow_interfaces_list                                | A list of strings, each identifying a range of interface addresses or an interface name. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface.  
  For example, the following are acceptable strings:  
  192.168.1.1  
  192.168.1.*  
  192.*  
  ether0  
  If the list is non-empty, this "white" list is applied before the parent.deny_interfaces_list on the next page list. The DomainParticipant will use the resulting list of interfaces to inform its remote participant(s) about which unicast addresses may be used to contact the DomainParticipant.  
  The resulting list restricts reception to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform).  
  You must manage the memory of the list. The memory may be freed after the DomainParticipant is deleted.  
  Default: empty list that represents all available interfaces |
### Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parent.deny_interfaces_list</code></td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of these interfaces. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. For example, the following are acceptable strings: 192.168.1.1, 192.168.1.<em>, 192.168.</em>, 192.*. ether0. This &quot;black&quot; list is applied after the <code>parent.allow_interfaces_list</code> on the previous page list and filters out the interfaces that should not be used for receiving data. The resulting list restricts reception to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform). You must manage the memory of the list. The memory may be freed after the <code>DomainParticipant</code> is deleted. Default: empty list that represents no denied interfaces.</td>
</tr>
<tr>
<td><code>parent.allow_multicast_interfaces_list</code></td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, allow the use of multicast only on these interfaces. If the list is empty, allow the use of all the allowed interfaces. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. This list sub-selects from the allowed interfaces that are obtained after applying the <code>parent.allow_interfaces_list</code> on the previous page &quot;white&quot; list and the <code>parent.deny_interfaces_list</code> above &quot;black&quot; list. From that resulting list, <code>parent.deny_multicast_interfaces_list</code> below is applied. Multicast output will be sent and may be received over the interfaces in the resulting list (if multicast is supported on the platform). If this list is empty, all the allowed interfaces may potentially be used for multicast. You must manage the memory of the list. The memory may be freed after the <code>DomainParticipant</code> is deleted. Default: empty list that represents all available interfaces.</td>
</tr>
<tr>
<td><code>parent.deny_multicast_interfaces_list</code></td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of those interfaces for multicast. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. This &quot;black&quot; list is applied after the <code>parent.allow_multicast_interfaces_list</code> above list and filters out the interfaces that should not be used for multicast. The final resulting list will be those interfaces that—if multicast is available—will be used for multicast sends. You must manage the memory of the list. The memory may be freed after the <code>DomainParticipant</code> is deleted. Default: empty list that represents no denied interfaces.</td>
</tr>
</tbody>
</table>
### 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy

#### Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv4.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>send_socket_buffer_size</td>
<td>Size in bytes of the send buffer of a socket used for sending. On most operating systems, setsockopt() will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to the property, parent.message_size_max on page 802. The maximum value is operating system-dependent. If -1, setsockopt() (or equivalent) will not be called to size the send buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td>recv_socket_buffer_size</td>
<td>Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to the property, parent.message_size_max on page 802. The maximum value is operating system-dependent. If -1, setsockopt() (or equivalent) will not be called to size the receive buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td>unicast_enabled</td>
<td>Allows the transport plugin to use unicast UDP for sending and receiving. By default, it will be turned on. Also by default, it will use all the allowed network interfaces that it finds up and running when the plugin is instanced. Can be 1 (enabled) or 0 (disabled). Default: 1</td>
</tr>
<tr>
<td>multicast_enabled</td>
<td>Allows the transport plugin to use multicast for sending and receiving. You can turn multicast on or off for this plugin. The default is that multicast is on and the plugin will use the all network interfaces allowed for multicast that it finds up and running when the plugin is instanced. Can be 1 (enabled) or 0 (disabled). Default: 1</td>
</tr>
<tr>
<td>multicast_ttl</td>
<td>Value for the time-to-live parameter for all multicast sends using this plugin. This is used to set the TTL of multicast packets sent by this transport plugin. Default: 1</td>
</tr>
<tr>
<td>multicast_loopback_disabled</td>
<td>Prevents the transport plugin from putting multicast packets onto the loopback interface. If disabled, then when sending multicast packets, do not put a copy on the loopback interface. This will prevent other applications on the same node (including itself) from receiving those packets. <strong>Note:</strong> Windows CE does not support multicast loopback. This field is ignored for Windows CE targets. Default: 0, meaning multicast loopback is enabled. Turning off multicast loopback (setting to 1) may result in minor performance gains when using multicast.</td>
</tr>
</tbody>
</table>
### Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignore_loopback_interface</td>
<td>Prevents the transport plugin from using the IP loopback interface. Three values are allowed:</td>
</tr>
<tr>
<td></td>
<td>- 0: Forces local traffic to be sent over loopback, even if a more efficient transport (such as shared memory) is installed (in which case traffic will be sent over both transports).</td>
</tr>
<tr>
<td></td>
<td>- 1: Disables local traffic via this plugin. The IP loopback interface will not be used, even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient transport (such as shared memory) instead of the IP loopback.</td>
</tr>
<tr>
<td></td>
<td>- -1: Automatic. Enables local traffic via this plugin. To avoid redundant traffic, Connext DDS will selectively ignore the loopback destinations that are also reachable through shared memory.</td>
</tr>
<tr>
<td>Default: -1</td>
<td></td>
</tr>
<tr>
<td>ignore_nonup_interfaces</td>
<td>This property is only supported on Windows platforms with statically configured IP addresses. It allows/disallows the use of interfaces that are not reported as UP (by the operating system) in the UDPv4 transport. Two values are allowed:</td>
</tr>
<tr>
<td></td>
<td>- 0: Allow interfaces that are reported as DOWN.</td>
</tr>
<tr>
<td></td>
<td>- Setting this value to 0 supports communication scenarios in which interfaces are enabled after the participant is created. Once the interfaces are enabled, discovery will not occur until the participant sends the next periodic announce-</td>
</tr>
<tr>
<td></td>
<td>- ment (controlled by the parameter $participant_qos.discovery_configparticipant_liveliness ASSERT_period$).</td>
</tr>
<tr>
<td></td>
<td>- To reduce discovery time, you may want to decrease the value of $participant_liveliness ASSERT_period$. For the above scenario, there is one caveat: non-UP interfaces must have a static IP assigned.</td>
</tr>
<tr>
<td></td>
<td>- 1: Do not allow interfaces that are reported as DOWN.</td>
</tr>
<tr>
<td>Default: 1</td>
<td></td>
</tr>
<tr>
<td>ignore_nonrunning_interfaces</td>
<td>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system.</td>
</tr>
<tr>
<td></td>
<td>The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that &quot;all resources are allocated&quot;, and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed:</td>
</tr>
<tr>
<td></td>
<td>- 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.</td>
</tr>
<tr>
<td></td>
<td>- 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.</td>
</tr>
<tr>
<td>By default this property is set to 1, so Connext DDS will ignore non-running interfaces.</td>
<td></td>
</tr>
</tbody>
</table>
### Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv4.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>no_zero_copy</td>
<td>Prevents the transport plugin from doing a zero copy. By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application. The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the hardware, device driver, or the OS to allow the zero-copy feature to work for your application, you may have no choice but to turn off zero-copy. By default this is set to 0, so Connext DDS will use the zero-copy API if offered by the OS.</td>
</tr>
</tbody>
</table>
| send_blocking | Controls the blocking behavior of send sockets. **CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.** Currently two values are defined:  
  - 1 (NDDS_TRANSPORT_UDP_BLOCKING_ALWAYS): Sockets are blocking (default socket options for operating system).  
  - 0 (NDDS_TRANSPORT_UDP_BLOCKING_NEVER): Sockets are modified to make them non-blocking. **This may cause significant performance problems.**  

  Default: 1 |
| transport_priority_mask | Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low below and transport_priority_mapping_high below to define the mapping from the 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458 to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket.  

  For example, the value 0x00000000 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xffffffff in this case) to the range specified by low and high.  

  If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets.  

  Default: 0 |
| transport_priority_mapping_low | Sets the low and high values of the output range to IPv4 TOS. These values are used in conjunction with transport_priority_mask above to define the mapping from the 7.5.26 TRANSPORT_PRIORITY QosPolicy on page 458 to the IPv4 TOS field. Defines the low and high values of the output range for scaling.  

  Note that IPv4 TOS is generally an 8-bit value.  

  Default: 0 for transport_priority_mapping_low and 0xFF for transport_priority_mapping_high |
| transport_priority_mapping_high | This property specifies whether to send a PING message before commencing the discovery process. On certain operating systems or with certain switches the initial UDP packet, configuring the ARP table, was unfortunately dropped. To avoid dropping the initial RTPS discovery sample, a PING message is sent to preconfigure the ARP table in those environments.  

  Default: 1 |
## Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv4.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>interface_poll_period</td>
<td>Specifies the period in milliseconds to query for changes in the state of all the interfaces. When possible, the detection of an IP address changes is done asynchronously using the APIs offered by the underlying OS. If there is no mechanism to do that, the detection will use a polling strategy where the polling period can be configured by setting this property. Default: 500</td>
</tr>
<tr>
<td>reuse_multicast_receive_resource</td>
<td>Controls whether or not to reuse receive resources. Setting this to 0 (FALSE) prevents multicast crosstalk by uniquely configuring a port and creating a receive thread for each multicast group address. Affects Linux systems only; ignored for non-Linux systems. Default: 1</td>
</tr>
<tr>
<td>protocol_overhead_max</td>
<td>Maximum size in bytes of protocol overhead, including headers. This value is the maximum size, in bytes, of protocol-related overhead. Normally, the overhead accounts for UDP and IP headers. The default value is set to accommodate the most common UDP/IP header size. Note that when parent.message_size_max on page 802 plus this overhead is larger than the UDPv4 maximum message size (65535 bytes), the middleware will automatically reduce the effective message_size_max to 65535 minus this overhead. Default: 28</td>
</tr>
<tr>
<td>disable_interface_tracking</td>
<td>Disables detection of network interface changes. By default, network interfaces changes are propagated in the form of locators to other applications. This is done to support IP mobility scenarios. For example, you could start an application with Wi-Fi and move to a wired connection. In order to continue communicating with other applications this interface change must be propagated. In 5.0 and earlier versions of the product, IP mobility scenarios were not supported. Applications using 5.2 will report errors if they detect locator changes in a DataWriter or DataReader. You can disable the notification and propagation of interface changes by setting this property to 1. This way, an interface change in a newer application will not trigger errors in an application running 5.2 GAR or earlier. Of course, this will prevent the new application from being able to detect network interface changes.</td>
</tr>
</tbody>
</table>
### 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy

#### Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
</table>
| public_address (prefix with ‘dds.transport.UDPv4.builtin.’) | - Public IP address associated with the transport instantiation.  
- Setting the public IP address is only necessary to support communication over WAN that involves Network Address Translation (NAT).  
- Typically, the address is the public address of the IP NAT router that provides access to the WAN.  
- By default, the DomainParticipant creating the transport will announce the IP addresses obtained from the NICs to other DomainParticipants in the system.  
- When this property is set, the DomainParticipant will announce the IP address corresponding to the property value instead of the LAN IP addresses associated with the NICs.  
- Notes:  
  - Setting this property is necessary, but is not a sufficient condition for sending and receiving data over the WAN. You must also configure the IP NAT router to allow UDP traffic and to map the public IP address specified by this property to the DomainParticipant’s private LAN IP address. This is typically done with one of these mechanisms:  
    - Port Forwarding: You must map the private ports used to receive discovery and user data traffic to the corresponding public ports (see Table 9.21 DDS_RtpsWellKnownPorts_t). Public and private ports must be the same since the transport does not allow you to change the mapping.  
    - 1:1 NAT: You must add a 1:1 NAT entry that maps the public IP address specified in this property to the private LAN IP address of the DomainParticipant.  
- By setting this property, the DomainParticipant only announces its public IP address to other DomainParticipants. Therefore, communication with DomainParticipants within the LAN that are running on different nodes will not work unless the NAT router is configured to enable NAT reflection (hairpin NAT).  
- There is another way to achieve simultaneous communication with DomainParticipants running in the LAN and WAN, that does not require hairpin NAT. This way uses a gateway application such as RTI Routing Service to provide access to the WAN.  
- Default: NULL (the transport uses the IP addresses obtained from the NICs) |
| use_checksum                                      | - This property specifies whether the UDP checksum will be computed. On Windows and Linux systems, the UDP checksum will not be set when use_checksum is set to 0. This is useful when RTPS protocol statistics related to corrupted messages need to be collected through the operation get_participant_protocol_status() (see 9.3.14 Getting Participant Protocol Status on page 631).  
- Default: 1 |
| force_interface_poll_detection                    | - This property forces the interface tracker to use a polling method to detect changes to the network interfaces in IP mobility scenarios. It only applies to operating systems that support asynchronous notifications of interface changes.  
- If set to TRUE, the interface tracker will use a polling method that queries the interfaces periodically to detect the changes. If set to FALSE, the interface tracker will use the operating system's default method.  
- Basically, this property allows you—for an operating system that supports asynchronous notification—to use the polling method instead.  
- Default: FALSE |
Table 16.2 Properties for the Builtin UDPv4 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv4.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>join_multicast_group_timeout</td>
<td>Windows only. On Windows, a network interface may be detected before it is allowed to join a multicast group address. This property adjusts how much time (in milliseconds) to wait for the ADD_MEMBERSHIP multicast operation to succeed before withdrawing. Default: 5000</td>
</tr>
</tbody>
</table>
| property_validation_action | By default, property names given in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the transport:  
  - VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail.  
  - VALIDATION_ACTION_SKIP: skip validation.  
  - VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail.  
If this property is not set, the property validation behavior will be the same as that of the DomainParticipant, which by default is VALIDATION_ACTION_EXCEPTION. See 7.5.19.1 Property Validation on page 441 for more information. |
| thread_name_prefix | You can set this field with your own value, to help you identify the transport thread in a way that's meaningful to you. Do not exceed 8 characters.  
If you do not set this field, Connext DDS creates the following prefix:  
  - ‘r’ + ‘Tr’ + participant identifier + ‘0’  
Where ‘r’ indicates this is a thread from RTI, ‘Tr’ indicates the thread is related to a transport, and participant identifier contains 5 characters as follows:  
  - If participant_name is set: The participant identifier will be the first 3 characters and the last 2 characters of the participant_name.  
  - If participant_name is not set, then the identifier is computed as domain_id (3 characters) followed by participant_id (2 characters).  
  - If participant_name is not set and the participant_id is set to -1 (default value), then the participant identifier is computed as the last 5 digits of the rtps_instance_id in the participant GUID.  
See 21.9 Identifying Threads Used by Connext DDS on page 916. |
### Table 16.3 Properties for Built-in UDPv6 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.UDPv6.builtin.')</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.properties_bitmap</td>
<td>A bitmap that defines various properties of the transport to the Connext DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connext DDS tries to receive a message using the plugin. This is in support of a zero-copy interface.</td>
</tr>
<tr>
<td>parent.gather_send_buffer_count_max</td>
<td>Specifies the maximum number of buffers that Connext DDS can pass to the send() method of a transport plugin. The transport plugin send() API supports a gather-send concept, where the send() call can take several discontiguous buffers, assemble and send them in a single message. This enables Connext DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer. However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connext DDS from trying to gather too many buffers into a send call for the transport plugin. Connext DDS requires all transport-plugin implementations to support a gather-send of at least a minimum number of buffers. This minimum number is NDDS_TRANSPORTPROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.</td>
</tr>
<tr>
<td>parent.message_size_max</td>
<td>The maximum size of a message in bytes that can be sent or received by the transport plugin. Above this size, DDS-level fragmentation will occur. See 23.3 Large Data Fragmentation on page 970. This value must be set before the transport plugin is registered, so that Connext DDS can properly use the plugin. Default for Integrity platforms: 9196 Default for non-Integrity platforms: 65487</td>
</tr>
<tr>
<td>parent.allow_interfaces_list</td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See 16.6.2 Formatting Rules for IPv6 'Allow' and 'Deny' Address Lists on page 820. If the list is non-empty, this &quot;white&quot; list is applied before the parent.deny_interfaces_list below list. The DomainParticipant will use the resulting list of interfaces to inform its remote participant(s) about which unicast addresses may be used to contact the DomainParticipant. The resulting list restricts reception to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform). You must manage the memory of the list. The memory may be freed after the DomainParticipant is deleted.</td>
</tr>
<tr>
<td>parent.deny_interfaces_list</td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of these interfaces. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See 16.6.2 Formatting Rules for IPv6 'Allow' and 'Deny' Address Lists on page 820. This &quot;black&quot; list is applied after the parent.allow_interfaces_list above list and filters out the interfaces that should not be used. The resulting list restricts reception to a particular set of interfaces for unicast UDP. Multicast output will still be sent and may be received over the interfaces in the list (if multicast is supported on the platform). You must manage the memory of the list. The memory may be freed after the DomainParticipant is deleted.</td>
</tr>
</tbody>
</table>
## Table 16.3 Properties for Built-in UDPv6 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with <code>.dds.transport.UDPv6.builtin.</code>)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent..allow_multicast_interfaces_list</td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, allow the use of multicast only these interfaces; otherwise allow the use of all the allowed interfaces. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See 16.6.2 Formatting Rules for IPv6 ‘Allow’ and ‘Deny’ Address Lists on page 820. This list sub-selects from the allowed interfaces that are obtained after applying the parent.allow_multicast_interfaces_list on the previous page “white” list and the parent.deny_multicast_interfaces_list on the previous page “black” list. Finally, the parent.deny_multicast_interfaces_list below is applied. Multicast output will be sent and may be received over the interfaces in the resulting list (if multicast is supported on the platform). If this list is empty, all the allowed interfaces may potentially be used for multicast. You must manage the memory of the list. The memory may be freed after the DomainParticipant is deleted.</td>
</tr>
<tr>
<td>parent.deny_multicast_interfaces_list</td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of those interfaces for multicast. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. See 16.6.2 Formatting Rules for IPv6 ‘Allow’ and ‘Deny’ Address Lists on page 820. This &quot;black&quot; list is applied after the parent.allow_multicast_interfaces_list above list and filters out the interfaces that should not be used for multicast. Multicast output will be sent and may be received over the interfaces in the resulting list (if multicast is supported on the platform). You must manage the memory of the list. The memory may be freed after the DomainParticipant is deleted.</td>
</tr>
<tr>
<td>send_socket_buffer_size</td>
<td>Size in bytes of the send buffer of a socket used for sending. On most operating systems, <code>setsockopt()</code> will be called to set the SENDBUFF to the value of this parameter. This value must be greater than or equal to <code>parent.message_size_max</code>. The maximum value is operating system-dependent. If -1, <code>setsockopt()</code> (or equivalent) will not be called to size the send buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td>recv_socket_buffer_size</td>
<td>Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, <code>setsockopt()</code> will be called to set the RECVBUFF to the value of this parameter. This value must be greater than or equal to <code>parent.message_size_max</code>. The maximum value is operating system-dependent. If -1, <code>setsockopt()</code> (or equivalent) will not be called to size the receive buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td>unicast_enabled</td>
<td>Allows the transport plugin to use unicast UDP for sending and receiving. By default, it will be turned on (1). Also by default, it will use all the allowed network interfaces that it finds up and running when the plugin is instanced. Can be 1 (enabled) or 0 (disabled).</td>
</tr>
</tbody>
</table>
### Table 16.3 Properties for Builtin UDPv6 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv6.builtin.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>multicast_enabled</strong></td>
<td>Allows the transport plugin to use multicast for sending and receiving. You can turn multicast UDP on or off for this plugin. By default, it will be turned on (1). Also by default, it will use the all network interfaces allowed for multicast that it finds up and running when the plugin is instanced. Can be 1 (enabled) or 0 (disabled).</td>
</tr>
<tr>
<td><strong>multicast_ttl</strong></td>
<td>Value for the time-to-live parameter for all multicast sends using this plugin. This is used to set the TTL of multicast packets sent by this transport plugin.</td>
</tr>
<tr>
<td><strong>multicast_loopback_disabled</strong></td>
<td>Prevents the transport plugin from putting multicast packets onto the loopback interface. If disabled, then when sending multicast packets, Connext DDS will not put a copy on the loopback interface. This will prevent applications on the same node (including itself) from receiving those packets. This is set to 0 by default, meaning multicast loopback is enabled. Disabling multicast loopback off (setting this value to 1) may result in minor performance gains when using multicast.</td>
</tr>
<tr>
<td><strong>ignore_loopback_interface</strong></td>
<td>Prevents the transport plugin from using the IP loopback interface. Three values are allowed: 0: Enable local traffic via this plugin. This plugin will only use and report the IP loopback interface if there are no other network interfaces (NICs) up on the system. 1: Disable local traffic via this plugin. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plugin like Shared Memory instead of the IP loopback. -1: Automatic. Enables local traffic via this plugin. To avoid redundant traffic, Connext DDS will selectively ignore the loopback destinations that are also reachable through shared memory.</td>
</tr>
<tr>
<td><strong>ignore_nonrunning_interfaces</strong></td>
<td>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that “all resources are allocated”, and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed: 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network. By default this property is set to 1, so Connext DDS will ignore non-running interfaces.</td>
</tr>
<tr>
<td><strong>DEPRECATED no_zero_copy</strong></td>
<td>Prevents the transport plugin from doing a zero copy. By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application. The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the H/W, device driver, or the OS to allow the zero-copy feature to work for your application, you may have no choice but to turn off zero-copy. By default this is set to 0, so Connext DDS will use the zero-copy API if offered by the OS.</td>
</tr>
</tbody>
</table>
### Table 16.3 Properties for Builtin UDPv6 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv6.builtin.’)</th>
<th>Description</th>
</tr>
</thead>
</table>
| send_blocking                                               | Controls the blocking behavior of send sockets. **CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.** Currently two values are defined:  
- 1 (NDDS_TRANSPORT_UDP_BLOCKING_ALWAYS): Sockets are blocking (default socket options for Operating System).  
- 0 (NDDS_TRANSPORT_UDP_BLOCKING_NEVER): Sockets are modified to make them non-blocking. **This may cause significant performance problems.**  
Default: 1 |
| enable_v4mapped                                             | Specifies whether the UDPv6 transport will process IPv4 addresses.  
Set this to 1 to turn on processing of IPv4 addresses. Note that this may make it incompatible with use of the UDPv4 transport within the same DomainParticipant. |
| transport_priority_mask                                     | Sets a mask for use of transport priority field.  
If transport priority mapping is supported on the platform\(^1\), this mask is used in conjunction with `transport_priority_mapping_low` and `transport_priority_mapping_high` to define the mapping from the DDS transport priority 7.5.26 TRANSPORT_PRIORITY qosPolicy on page 458 to the IPv6 TCLASS field.  
Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv6 TCLASS field on an outgoing socket.  
For example, the value 0x00000ff0 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high.  
If the mask is set to zero, then the transport will not set IPv6 TCLASS for send sockets. |
| transport_priority_mapping_low                              | Sets the low and high values of the output range to IPv6 TCLASS.  
These values are used in conjunction with `transport_priority_mask` above to define the mapping from DDS transport priority to the IPv6 TCLASS field. Defines the low and high values of the output range for scaling.  
Note that IPv6 TCLASS is generally an 8-bit value. |
| transport_priority_mapping_high                             | |
| send_ping                                                   | This property specifies whether to send a PING message before commencing the discovery process. On certain operating systems or with certain switches the initial UDP packet, configuring the ARP table, was unfortunately dropped. To avoid dropping the initial RTPS discovery sample, a PING message is sent to preconfigure the ARP table in those environments. |
| interface_poll_period                                       | See [interface_poll_period on page 807](#) in Table 16.2 Properties for the Builtin UDPv4 Transport |
| reuse_multicast_receive_resource                            | This property controls whether or not to reuse multicast receive resources. |

---

\(^1\)See the [RTI Connext DDS Core Libraries Platform Notes](#) to find out if the transport priority is supported on a specific platform.
### Table 16.3 Properties for Builtin UDPv6 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv6.builtin.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protocol_overhead_max</td>
<td>This value is the maximum size, in bytes, of protocol-related overhead. Normally, the overhead accounts for UDP and IP headers. The default value is set to accommodate the most common UDP/IP header size. Note that when <code>NDDS_Transport_Property_t::message_size_max</code> plus this overhead is larger than the <code>parent.message_size_max</code> on page 810 (65535 bytes), the middleware will automatically reduce the effective <code>message_size_max</code>, to 65535 minus this overhead.</td>
</tr>
<tr>
<td>disable_interface_tracking</td>
<td>Disables detection of network interface changes. See <code>disable_interface_tracking</code> in Table 16.2 Properties for the Builtin UDPv4 Transport.</td>
</tr>
<tr>
<td>public_address</td>
<td>See <code>public_address</code> in Table 16.2 Properties for the Builtin UDPv4 Transport.</td>
</tr>
<tr>
<td>force_interface_poll_detection</td>
<td>This property forces the interface tracker to use a polling method to detect changes to the network interfaces in IP mobility scenarios. It only applies to operating systems that support asynchronous notifications of interface changes. If set to TRUE, the interface tracker will use a polling method that queries the interfaces periodically to detect the changes. If set to FALSE, the interface tracker will use the operating system's default method. Basically, this property allows you—for an operating system that supports asynchronous notification—to use the polling method instead. Default: FALSE</td>
</tr>
<tr>
<td>join_multicast_group_timeout</td>
<td>Windows only. On Windows, a network interface may be detected before it is allowed to join a multicast group address. This property adjusts how much time (in milliseconds) to wait for the ADD_MEMBERSHIP multicast operation to succeed before withdrawing. Default: 5000</td>
</tr>
</tbody>
</table>
| property_validation_action | By default, property names given in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the transport:  

- **VALIDATION_ACTION_EXCEPTION**: validate the properties. Upon failure, log errors and fail.  
- **VALIDATION_ACTION_SKIP**: skip validation.  
- **VALIDATION_ACTION_WARNING**: validate the properties. Upon failure, log warnings and do not fail.  

If this property is not set, the property validation behavior will be the same as that of the DomainParticipant, which by default is VALIDATION_ACTION_EXCEPTION. See 7.5.19.1 Property Validation on page 441 for more information. |
### 16.6 Setting Built-in Transport Properties with the Property QosPolicy

Table 16.3 Properties for Built-in UDPv6 Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.UDPv6.builtin.’)</th>
<th>Description</th>
</tr>
</thead>
</table>
| thread_name_prefix | You can set this field with your own value, to help you identify the transport thread in a way that's meaningful to you. Do not exceed 8 characters. If you do not set this field, Connext DDS creates the following prefix: ‘r’ + ‘Tr’ + participant identifier + ‘0’ Where ‘r’ indicates this is a thread from RTI, ‘Tr’ indicates the thread is related to a transport, and participant identifier contains 5 characters as follows:  
  - If `participant_name` is set: The participant identifier will be the first 3 characters and the last 2 characters of the `participant_name`.  
  - If `participant_name` is not set, then the identifier is computed as `domain_id` (3 characters) followed by `participant_id` (2 characters).  
  - If `participant_name` is not set and the `participant_id` is set to -1 (default value), then the participant identifier is computed as the last 5 digits of the `rtps_instance_id` in the participant GUID.  
See 21.9 Identifying Threads Used by Connext DDS on page 916. |

Table 16.4 Properties for Built-in Shared-Memory Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.shmem.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.properties_bitmap</td>
<td>A bitmap that defines various properties of the transport to the Connext DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connext DDS tries to receive a message using the plugin. This is in support of a zero-copy interface.</td>
</tr>
<tr>
<td>parent.gather_send_buffer_count_max</td>
<td>Specifies the maximum number of buffers that Connext DDS can pass to the <code>send()</code> method of a transport plugin. The transport plugin <code>send()</code> API supports a gather-send concept, where the <code>send()</code> call can take several discontiguous buffers, assemble and send them in a single message. This enables Connext DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer. However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connext DDS from trying to gather too many buffers into a send call for the transport plugin. Connext DDS requires all transport-plugin implementations to support a gather-send of at least a minimum number of buffers. This minimum is NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.</td>
</tr>
</tbody>
</table>
### Table 16.4 Properties for Builtin Shared-Memory Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with <code>dds.transport.shmem.builtin</code>)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>parent.message_size_max</code></td>
<td>The maximum size of a message in bytes that can be sent or received by the transport plugin. Above this size, DDS-level fragmentation will occur. See 23.3 Large Data Fragmentation on page 970. This value must be set before the transport plugin is registered, so that Connext DDS can properly use the plugin. Default for Integrity platforms: 9216 Default for non-Integrity platforms: 65536</td>
</tr>
<tr>
<td><code>parent.allow_interfaces_list</code></td>
<td>Not applicable to the Shared-Memory Transport</td>
</tr>
<tr>
<td><code>parent.deny_interfaces_list</code></td>
<td></td>
</tr>
<tr>
<td><code>parent.allow_multicast_interfaces_list</code></td>
<td></td>
</tr>
<tr>
<td><code>parent.deny_multicast_interfaces_list</code></td>
<td></td>
</tr>
<tr>
<td><code>received_message_count_max</code></td>
<td>Number of messages that can be buffered in the receive queue. This is the maximum number of messages that can be buffered in a RecvResource of the Transport Plugin. This does not guarantee that the Transport-Plugin will actually be able to buffer <code>received_message_count_max</code> messages of the maximum size set in <code>parent.message_size_max</code> above. The total number of bytes that can be buffered for a RecvResource is actually controlled by <code>receive_buffer_size</code> on the next page.</td>
</tr>
</tbody>
</table>

816
### Table 16.4 Properties for Builtin Shared-Memory Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with <code>dds.transport.shmem.builtin.</code>)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>receive_buffer_size</td>
<td>The total number of bytes that can be buffered in the receive queue.</td>
</tr>
<tr>
<td></td>
<td>This number controls how much memory is allocated by the plugin for the receive queue (on a per RecvRe-</td>
</tr>
<tr>
<td></td>
<td>source basis). The actual number of bytes allocated is:</td>
</tr>
</tbody>
</table>
|                                                          | \[
|                                                          | \text{size} = \text{receive_buffer_size} + \text{message_size_max} + \text{received_message_count_max} \times \text{fixedOverhead} |
|                                                          | \] |
|                                                          | where \text{fixedOverhead} is some small number of bytes used by the queue data structure. |
|                                                          | If \text{receive_buffer_size} < (\text{message_size_max} \times \text{received_message_count_max}), the transport plugin will not be able to store |
|                                                          | \text{received_message_count_max} messages of size \text{message_size_max}. |
|                                                          | If \text{receive_buffer_size} > (\text{message_size_max} \times \text{received_message_count_max}), then there will be memory allocated that cannot |
|                                                          | be used by the plugin and thus wasted. |
|                                                          | To optimize memory usage, specify a receive queue size less than that required to hold the maximum |
|                                                          | number of messages which are all of the maximum size. |
|                                                          | In most situations, the average message size may be far less than the maximum message size. So for ex- |
|                                                          | ample, if the maximum message size is 64K bytes, and you configure the plugin to buffer at least 10 mes- |
|                                                          | sages, then 640K bytes of memory would be needed if all messages were 64K bytes. Should this be |
|                                                          | desired, then \text{receive_buffer_size} should be set to 640K bytes. |
|                                                          | However, if the average message size is only 10K bytes, then you could set the \text{receive_buffer_size} to |
|                                                          | 100K bytes. This allows you to optimize the memory usage of the plugin for the average case and yet al- |
|                                                          | low the plugin to handle the extreme case. |
|                                                          | The queue will always be able to hold 1 message of \text{message_size_max} bytes, regardless of the value |
|                                                          | of \text{receive_buffer_size}. |
| host_id                                                  | Host ID used to generate the shared memory transport network address. |
|                                                          | Shared memory transport has an associated network address to communicate with other \text{DomainParticipants} within the same node. This network address is typically generated from the host ID, a |
|                                                          | unique host identifier. Connexx DDS computes this host ID based on the hardware address, or media ac- |
|                                                          | cess control (MAC) address, of the first network interface found and the value of \text{rtps_auto_id_kind}. |
|                                                          | When set, this property forces the use of a specific host ID to generate the shared memory network ad- |
|                                                          | dress instead of computing it as described above. This property takes an unsigned integer value that is |
|                                                          | converted into the network address. |
|                                                          | This host ID should satisfy the following properties: |
|                                                          | - Should be unique across nodes. Otherwise, remote \text{DomainParticipants} may try to com- |
|                                                          | municate using shared memory transport with \text{DomainParticipants} from a different node (which will not work). |
|                                                          | - Should be the same for all \text{DomainParticipants} within the same node that want to communicate |
|                                                          | using shared memory if \text{accept_unknown_peers} is set to FALSE or there are \text{DomainPar- |
|                                                          | ticipants} in the system running a Connexx DDS version previous to 6.0.0. |
|                                                          | **Note:** This property is needed in very few scenarios: for example, when two different Connexx DDS ap- |
|                                                          | plications in the same node have \text{rtps_auto_id_kind} set to DDS\_RTPS\_AUTO\_ID\_FROM\_UUID, the |
|                                                          | first detected network interface is different for each application, and \text{accept_unknown_peers} is set to |
|                                                          | FALSE.
### Table 16.4 Properties for Builtin Shared-Memory Transport

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.shmem.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>enable_udp_debugging</td>
<td>Enables UDP debugging when using shared memory. If set to '1', all shared memory traffic will be published to <code>udp_debugging_address::udp_debugging_port</code>, and the number of shared memory transport gather buffers will be the value of <code>parent.gather_send_buffer_count_max</code> or 16, whichever is smaller. Default: 0.</td>
</tr>
<tr>
<td>udp_debugging_address</td>
<td>IP address to which shared memory traffic will be published if <code>enable_udp_debugging</code> is set to ‘1’. Default: 239.255.1.2.</td>
</tr>
<tr>
<td>udp_debugging_port</td>
<td>Port to which shared memory traffic will be published if <code>enable_udp_debugging</code> is set to ‘1’. Default: 7399.</td>
</tr>
<tr>
<td>use_530_from_uuid_locator</td>
<td>This property only applies when the WireProtocol QoS policy (specifically <code>rtps_auto_id_kind</code>) is set to <code>DSS_RTPS_AUTO_ID_FROM_UUID</code>. If set to TRUE, the generated shared memory locator will be compatible with the locator created in version 5.3. If set to FALSE the generated shared memory locator will not be compatible, and communication will not occur. For more information, see the Migration Guide on the RTI Community Portal (<a href="https://community.rti.com/documentation">https://community.rti.com/documentation</a>). Default: FALSE</td>
</tr>
</tbody>
</table>
| property_validation_action                                  | By default, property names given in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the transport:  
  - VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail.  
  - VALIDATION_ACTION_SKIP: skip validation.  
  - VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail.  
  
  If this property is not set, the property validation behavior will be the same as that of the DomainParticipant, which by default is VALIDATION_ACTION_EXCEPTION. See 7.5.19.1 Property Validation on page 441 for more information. |
| thread_name_prefix                                          | You can set this field with your own value, to help you identify the transport thread in a way that's meaningful to you. Do not exceed 8 characters. If you do not set this field, Connext DDS creates the following prefix:  
  `y + 'T' + participant identifier + "0"`  
  Where ‘y’ indicates this is a thread from RTI, ‘T’ indicates the thread is related to a transport, and participant identifier contains 5 characters as follows:  
  - If `participant_name` is set: The participant identifier will be the first 3 characters and the last 2 characters of the `participant_name`.  
  - If `participant_name` is not set, then the identifier is computed as `domain_id` (3 characters) followed by `participant_id` (2 characters).  
  - If `participant_name` is not set and the `participant_id` is set to -1 (default value), then the participant identifier is computed as the last 5 digits of the `rtps_instance_id` in the participant GUID.  
  
  See 21.9 Identifying Threads Used by Connext DDS on page 916. |
16.6.1 Setting the Maximum Gather-Send Buffer Count for UDP Transports

To minimize memory copies, *Connext DDS* uses the "gather send" API that may be available on the transport.

Some operating systems limit the number of gather buffers that can be given to the gather-send function. This limits *Connext DDS's* ability to concatenate multiple DDS samples into a single network message. An example is the UDP transport's `sendmsg()` call, which on some OSs (such as Solaris) can only take 16 gather buffers, limiting the number of DDS samples that can be concatenated to five or six.

To match this limitation, *Connext DDS* sets the UDP transport plug-ins' `gather_send_buffer_count_max` to 16 by default for all operating systems. This field is part of the `NDDS_Transport_Property_t` structure:

- On VxWorks 5.5 operating systems, `gather_send_buffer_count_max` can be set as high as 63.
- On Windows and INTEGRITY operating systems, `gather_send_buffer_count_max` can be set as high as 128.
- On most other operating systems, `gather_send_buffer_count_max` can be set as high as 16.

If you are using an OS that allows more than 16 gather buffers for a `sendmsg()` call, you may increase the UDP transport plug-in's `gather_send_buffer_count_max` from the default up to your OS's limit (but no higher than 128).

For example, if your OS imposes a limit of 64 gather buffers, you may increase the `gather_send_buffer_count_max` up to 64. However, if your OS's gather-buffer limit is 1024, you may only increase the `gather_send_buffer_count_max` up to 128.

By changing `gather_send_buffer_count_max`, you can increase performance in the following situations:

- When a *DataWriter* is sending multiple packets to a *DataReader* either because the *DataReader* is a late-joiner and needs to catch up, or because several packets were dropped by the network or rejected and need to be resent. Changing the setting will help when the *DataWriter* needs to send or resend more than five or six packets at a time.
- If your application has more than five or six *DataWriters* or *DataReaders* in a participant. (In this case, the change will make the discovery process more efficient.)
- When using an asynchronous *DataWriter*, DDS samples are sent asynchronously by a separate thread. DDS samples may not be sent immediately, but may be queued instead, depending on the settings of the associated FlowController. If multiple DDS samples in the queue must be sent to the same destination, they will be coalesced into as few network packets as possible. The number of DDS samples that can be put in a single message is directly proportional to `gather_send_buffer_count_max`. Therefore, by maximizing `gather_send_buffer_count_max`, you can minimize the number of packets on the wire.
16.6.2 Formatting Rules for IPv6 ‘Allow’ and ‘Deny’ Address Lists

This section describes how to format the strings in the properties that create “allow” and “deny” lists:

- dds.transport.UDPv6.builtin. parent.allow_interfaces_list on page 802
- dds.transport.UDPv6.builtin. parent.deny_interfaces_list on page 803
- dds.transport.UDPv6.builtin. parent.allow_multicast_interfaces_list on page 803
- dds.transport.UDPv6.builtin. parent.deny_multicast_interfaces_list on page 803

These properties may contain a list of strings, each identifying a range of interface addresses or an interface name. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface.

The strings can be addresses and patterns in IPv6 notation. They are case-insensitive.

They may contain a wildcard ‘*’ and can expand up to 4 digits in a block. The wildcard must be either leading or trailing (cannot be in the middle of the string). Multiple wildcards can be specified in a single filter, but only one wildcard can be specified per block (between colons). Table 16.5 Examples of IPv6 Address Filters shows some examples.

### Table 16.5 Examples of IPv6 Address Filters

<table>
<thead>
<tr>
<th>Example Filter</th>
<th>Equivalent Filters</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>::::<em>::</em>:*</td>
<td>fe80::<em>:</em>:*</td>
<td>FE80:0000:0000:0000:0000:0000:xxxx:xxxx</td>
</tr>
<tr>
<td>FF00:0:::*</td>
<td>Fe80:0000:0000:0000:0000:0000:xxxx:xxxx</td>
<td></td>
</tr>
<tr>
<td>FE80:aBC::2*:*:2</td>
<td>FE80:0000:0000:0000:0000:0202:2xxx:xxxx</td>
<td></td>
</tr>
</tbody>
</table>

16.6.3 RTPS Overhead

Connext DDS adds protocol information to every RTPS message it sends out, and, by default, it reserves 512 bytes for this information. If the transport property parent.message_size_max (e.g., dds.transport.UDPv4.builtin.parent.message_size_max) is set to a small value such as 1400 bytes, the amount of overhead reserved per RTPS message may become large relative to the available space for user payload, leading to inefficient bandwidth utilization. The property dds.participant.protocol.rtps_overhead (see 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439) is used to adjust the overhead of the RTPS protocol without including security. The security overhead added by enabling sub-message or message encryption is automatically added to the dds.participant.protocol.rtps_overhead value.

The maximum user payload per data packet is (parent.message_size_max - rtps_overhead - security_rtps_overhead).
16.7 Installing Additional Builtin Transport Plugins with register_transport()

After you create an instance of a transport plugin (see 16.4 Explicitly Creating Builtin Transport Plugin Instances on page 798), you have to register it.

The builtin transports (UDPv4, UDPv6, and Shared Memory) are implicitly registered by default (if they are enabled via the 9.5.7 TRANSPORT_BUILTIN QosPolicy (DDS Extension) on page 665). Therefore, you only need to explicitly register a builtin transport if you want an extra instance of it (suppose you want two UDPv4 transports, one with special settings).

The register_transport() operation registers a transport plugin for use with a DomainParticipant and assigns it a network address. (Note: this operation is only available in the APIs other than Java or .NET. If you are using Java or .NET, use the Property QosPolicy to install additional transport plugins.)

```c
NDDS_Transport_Handle_t NDDSTransportSupport::register_transport(
    DDSDomainParticipant * participant_in,
    NDDS_Transport_Plugin * transport_in,
    const DDS_StringSeq & aliases_in,
    const NDDS_Transport_Address_t & network_address_in)
```

Where:

- **participant_in** A non-NULL, disabled DomainParticipant.
- **transport_in** A non-NULL transport plugin that is currently not registered with another DomainParticipant.
- **aliases_in** A non-NULL sequence of strings used as aliases to refer to the transport plugin symbolically. The transport plugin will be "available for use" to an Entity contained in the DomainParticipant, if the transport alias list associated with the Entity contains one of these transport aliases. An empty alias list represents a WILDCARD and matches ALL aliases. See 16.7.2 Transport Aliases on the next page.
- **network_address_in** The network address at which to register this transport plugin. The least significant transport_in-.property.address_bit_count will be truncated. The remaining bits are the network address of the transport plugin. See 16.7.3 Transport Network Addresses on page 823.

**Note:** You must ensure that the transport plugin instance is only used by one DomainParticipant at a time. See 16.7.1 Transport Lifecycles on the next page.

Upon success, a valid non-NIL transport handle is returned, representing the association between the DomainParticipant and the transport plugin. If the transport cannot be registered, NDDS_TRANSPORT_HANDLE_NIL is returned.

Note that a transport plugin's class name is automatically registered as an implicit alias for the plugin. Thus, a class name can be used to refer to all the transport plugin instances of that class.

The C and C++ APIs also have a operation to retrieve a registered transport plugin, get_transport_plugin().
16.7.1 Transport Lifecycles

If you create and register a transport plugin with a DomainParticipant, you are responsible for deleting it by calling its destructor. Built-in transport plugins are automatically managed by Connext DDS if they are implicitly registered through the TransportBuiltInQosPolicy.

User-created transport plugins must not be deleted while they are is still in use by a DomainParticipant. This generally means that a user-created transport plugin instance can only be deleted after the DomainParticipant with which it was registered is deleted. Note that a transport plugin cannot be "unregistered" from a DomainParticipant.

A transport plugin instance cannot be registered with more than one DomainParticipant at a time. This requirement is necessary to guarantee the multi-threaded safety of the transport API.

Thus, if the same physical transport resources are to be used with multiple DomainParticipants in the same address space, the transport plugin should be written in such a way so that it can be instantiated multiple times—one for each DomainParticipant in the address space. Note that it is always possible to write the transport plugin so that multiple transport plugin instances share the same underlying resources; however the burden (if any) of guaranteeing multi-threaded safety to access shared resource shifts to the transport plugin developer.

16.7.2 Transport Aliases

In order to use a transport plugin instance in a Connext DDS application, it must be registered with a DomainParticipant using the register_transport() operation (16.7 Installing Additional Built-in Transport Plugins with register_transport() on the previous page). register_transport() takes a pointer to the transport plugin instance, and in addition allows you to specify a sequence of "alias" strings to symbolically refer to the transport plugin. The same alias strings can be used to register more than one transport plugin.

Multiple transport plugins can be registered with a DomainParticipant. An alias symbolically refers to one or more transport plugins registered with the DomainParticipant. Pre-configured built-in transport plugin instances can be referred to using preconfigured aliases.

A transport plugin's class name is automatically used as an implicit alias. It can be used to refer to all the transport plugin instance of that class.

You can use aliases to refer to transport plugins in order to specify:

- Transport plugins to use for discovery (see enabled_transports in 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640), and for DataWriters and DataReaders (see 7.5.27 TRANSPORT_SELECTION QosPolicy (DDS Extension) on page 459).
- Multicast addresses on which to receive discovery messages (see `multicast_receive_addresses` in 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640), and the multicast addresses and ports on which to receive user data (DDS_DataReaderQos::multicast).
- Unicast ports used for user data (see 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460) on both DataWriters and DataReaders.
- Transport plugins used to parse an address string in a locator.

A DomainParticipant (and its contained entities) will start using a transport plugin after the DomainParticipant is enabled (see 4.1.2 Enabling DDS Entities on page 168). An entity will use all the transport plugins that match the specified transport QoS policy. All transport plugins are treated uniformly, regardless of how they were created or registered; there is no notion of some transports being more "special" than others.

### 16.7.3 Transport Network Addresses

The address bits *not* used by the transport plugin for its internal addressing constitute its network address bits.

In order for Connext DDS to properly route the messages, each unicast interface in the DDS domain must have a unique address.

You specify the network address when installing a transport plugin via the `register_transport()` operation (16.7 Installing Additional Builtin Transport Plugins with register_transport() on page 821). Choose the network address for a transport plugin so that the resulting fully qualified 128-bit address will be unique in the DDS domain.

If two instances of a transport plugin are registered with a DomainParticipant, they need different network addresses so that their unicast interfaces will have unique, fully qualified 128-bit addresses.

While it is possible to create multiple transports with the same network address (this can be useful for certain situations), this requires special entity configuration for most transports to avoid clashes in resource use (e.g., sockets for UDPv4 transport).

### 16.8 Installing Additional Builtin Transport Plugins with PropertyQosPolicy

Similar to default builtin transport instances, additional builtin transport instances can also be configured through 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.

To install additional instances of builtin transport, the Properties listed in Table 16.6 Properties for Dynamically Loading and Registering Additional Builtin Transport Plugins are required.
### Table 16.6 Properties for Dynamically Loading and Registering Additional Builtin Transport Plugins

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.transport.load_plugins</td>
<td>Comma-separated list of <code>&lt;TRANSPORT_PREFIX&gt;</code>. Up to 8 entries may be specified.</td>
</tr>
</tbody>
</table>
| `<TRANSPORT_PREFIX>`                 | Indicates the additional builtin transport instances to be installed, and must be in one of the following form, where `<STRING>` can be any string other than "builtin": dds.transport.shmem.<STRING>
|                                      | dds.transport.UDPv4.<STRING>                                                                                                               |
|                                      | dds.transport.UDPv6.<STRING>                                                                                                               |
|                                      | In the following examples in this table, `<TRANSPORT_PREFIX>` is used to indicate one element of this string that is used as a prefix in the property names for all the settings that are related to the plugin. |
| `<TRANSPORT_PREFIX>`, aliases         | Optional. Aliases used to register the transport to the DomainParticipant. Refer to the `aliases_in` parameter in `register_transport()` (see 16.7 Installing Additional Builtin Transport Plugins with `register_transport()` on page 821). Aliases should be specified as a comma-separated string, with each comma delimiting an alias. If it is not specified, the prefix—without the leading "dds.transport"—is used as the default alias for the plugin. For example, if the `<TRANSPORT_PREFIX>` is "dds.transport.mytransport", the default alias for the plugin is "mytransport". |
| `<TRANSPORT_PREFIX>`, network_address | Optional. Network address used to register the transport to the DomainParticipant. Refer to `network_address_in` parameter in `register_transport()` (see 16.7 Installing Additional Builtin Transport Plugins with `register_transport()` on page 821). If it is not specified, the `network_address_out` output parameter from NDDS_Transport_create_plugin is used. The default value is a zeroed out network address. |
| `<TRANSPORT_PREFIX>`, <property_name> | Optional. Property for creating the transport plugin. More than one `<TRANSPORT_PREFIX>`, `<property_name>` can be specified. See Table 16.2 Properties for the Builtin UDPv4 Transport through Table 16.4 Properties for Builtin Shared-Memory Transport for the property names that can be used to configure the additional builtin transport instances. The only difference is that the property name will be prefixed by `dds.transport.<builtin_transport_name>`, `<instance_name>`, where `<instance_name>` is configured through the `dds.transport.load_plugins` property instead of `dds.transport.<builtin_transport_name>.builtin`. |

### 16.9 Other Transport Support Operations

#### 16.9.1 Adding a Send Route

By default, a transport plugin will send outgoing messages using the network address range at which the plugin was registered.

The `add_send_route()` operation allows you to control the routing of outgoing messages, so that a transport plugin will only send messages to certain ranges of destination addresses.

Before using this operation, the `DomainParticipant` to which the transport is registered must be disabled.

```c
DDS_ReturnCode_t NDDSTransportSupport::add_send_route(
    const NDDS_Transport_Handle_t & transport_handle_in,
```
16.9.2 Adding a Receive Route

By default, a transport plugin will receive incoming messages using the network address range at which the plugin was registered.

The add_receive_route() operation allows you to configure a transport plugin so that it will only receive messages on certain ranges of addresses.

Before using this operation, the DomainParticipant to which the transport is registered must be disabled.

```cpp
const NDDS_Transport_Handle_t & transport_handle_in,
const NDDS_Transport_Address_t & address_range_in,
DDS_Long address_range_bit_count_in)
```

Where:

- **transport_handle_in**: A valid non-NIL transport handle as a result of a call to register_transport() (16.7 Installing Additional Builtin Transport Plugins with register_transport() on page 821).
- **address_range_in**: The incoming address range for which to use this transport plugin.
- **address_range_bit_count_in**: The number of most significant bits used to specify the address range.

It returns one of the standard return codes or DDS_RETCODE_PRECONDITION_NOT_MET.

The method can be called multiple times for a transport plugin, with different address ranges.

<table>
<thead>
<tr>
<th>Outgoing Address Range 1</th>
<th>-&gt;</th>
<th>Transport Plugin</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
<tr>
<td>Outgoing Address Range K</td>
<td>-&gt;</td>
<td>Transport Plugin</td>
</tr>
</tbody>
</table>

16.9.2 Adding a Receive Route

By default, a transport plugin will receive incoming messages using the network address range at which the plugin was registered.

The add_receive_route() operation allows you to configure a transport plugin so that it will only receive messages on certain ranges of addresses.

Before using this operation, the DomainParticipant to which the transport is registered must be disabled.

```cpp
const NDDS_Transport_Address_t & address_range_in,
DDS_Long address_range_bit_count_in)
```

Where:

- **transport_handle_in**: A valid non-NIL transport handle as a result of a call to register_transport() (16.7 Installing Additional Builtin Transport Plugins with register_transport() on page 821).
- **address_range_in**: The incoming address range for which to use this transport plugin.
- **address_range_bit_count_in**: The number of most significant bits used to specify the address range.

It returns one of the standard return codes or DDS_RETCODE_PRECONDITION_NOT_MET.

The method can be called multiple times for a transport plugin, with different address ranges.

| Transport Plugin | <- | Incoming Address Range 1 |
You can set up a routing table to restrict the use of a transport plugin to receive messages from selected ranges. For example, you may restrict a transport plugin to:

- Receive messages from a certain multicast address range.
- Receive messages only on certain unicast interfaces (when multiple unicast interfaces are available on the transport plugin).

### 16.9.3 Looking Up a Transport Plugin

If you need to get the handle associated with a transport plugin that is registered with a `DomainParticipant`, use the `lookup_transport()` operation.

```c
NDDS_Transport_Handle_t NDDS_TransportSupport::lookup_transport(
    DDSDomainParticipant * participant_in,
    DDS_StringSeq & aliases_out,
    NDDS_Transport_Address_t & network_address_out,
    NDDS_Transport_Plugin * transport_in )
```

Where:

- **participant_in**: A non-NULL `DomainParticipant`.
- **aliases_out**: A sequence of strings where the aliases used to refer to the transport plugin symbolically will be returned. NULL if not interested.
- **network_address_out**: The network address at which to register the transport plugin will be returned here. NULL if not interested.
- **transport_in**: A non-NULL transport plugin that is already registered with the `DomainParticipant`.

If successful, this operation returns a valid non-NIL transport handle, representing the association between the `DomainParticipant` and the transport plugin; otherwise it returns a `NDDS_TRANSPORT_HANDLE_NIL` upon failure.
Chapter 17 RTPS Locators and IP Mobility

DDS endpoints (DataWriters and DataReaders) can be reached at specific addresses called RTPS locators. An RTPS locator is an n-tuple (transport, address, port). For example (UDPv4, 192.168.1.1, 7400) is a locator for the UDPv4 transport. Locator information is sent as part of the Participant and Endpoint DATA messages (see Discovery (Chapter 15 on page 764)).

The initial set of locators that a DomainParticipant will use to communicate with other DomainParticipants is provided using a peer descriptor (see 15.2 Configuring the Peers List Used in Discovery on page 767).

17.1 Locator Changes at Run Time

In Connext DDS 5.2.3 and earlier, the set of locators associated with a DDS endpoint could not be changed after the DomainParticipant containing the endpoints was enabled. Therefore, Connext DDS was not prepared to deal with, for example, IP address changes in IP-based transports.

Starting with Connext DDS 5.3.0, locator changes are propagated as part of new Participant and Endpoint DATA messages.

17.1.1 Locator Changes in IP-Based Transports

For IP-based transports, including UDPv4 and UDPv6, the following IP mobility use cases (i.e., the need for IP-address changes at runtime) are supported in Connext DDS 5.3.0 and higher:

- Starting a DomainParticipant without network connectivity and connecting to the network at runtime.
- Switching network interfaces (for example, going from wired to Wi-Fi).
- Acquiring a new IP address after DHCP lease expiration.
- Having mobile devices roaming across network segments.

Connext DDS 5.3.0 introduced support for IP mobility for the following transports:
17.1.1 Locator Changes in IP-Based Transports

- UDPv4 and DTLSv4
- UDPv6
- TCPv4 and TLSv4
- LBRTPS
- ZRTPS

The functionality is enabled out-of-the-box.

When possible, the detection of IP address changes is done asynchronously using the APIs offered by the underlying OS. If there is no mechanism to do that, the detection will use a polling strategy.

The polling period can be configured using the following transport property in the `DomainParticipant's PropertyQosPolicy: <<transport prefix>>.interface_poll_period`. For example, for UDPv4 the property name is `dds.transport.UDPv4.builtin.interface_poll_period`.

17.1.1.1 Starting a DomainParticipant without Enabled Network Interfaces

For this use case, the GUID prefix generation algorithm must not be based on the IPv4/MAC address of the first enabled interface, but must use a UUID algorithm instead. This is necessary to avoid collisions on the GUID, which needs to be unique on the network.

In Connext DDS 5.3.x, to enable the use of a UUID algorithm to generate the GUID, you had to modify the `rtps_auto_id_kind` field in the `DomainParticipant's WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670` to `DDS_RTPS_AUTO_ID_FROM_UUID`. Starting with Connext DDS 6.0.0, the default value of this field changed to `DDS_RTPS_AUTO_ID_FROM_UUID`, and you do not need to modify it.

17.1.1.2 Locator Changes in IP-Based Transports when NATs are Involved

Locator changes at run time are not supported for UDP communications in the presence of NATs because this functionality is currently not supported by the RTI Secure WAN Transport.

For TCP communication, locator changes are supported on the client side in the presence of NATs as long as the TCP transport is used in asymmetric mode.

17.1.1.3 Disabling IP Locator Change Propagation

Connext DDS 5.2.3 and earlier will report errors if it detects locator changes in a DDS endpoint. You can disable the notification and propagation of these changes for a `DomainParticipant`. This way, an interface change in a 5.3.0 or higher application will not trigger errors in an application running 5.2.3 or earlier. Setting this property to true will prevent a 5.3.0 application from being able to detect network interface changes.
17.2 Detection of Unreachable Locators

To disable the notification of IP locator changes, set the following transport property in the DomainParticipant's PropertyQosPolicy: <<transport prefix>>.disable_interface_tracking. For example, for UDPv4 the property name is dds.transport.UDPv4.builtin.disable_interface_tracking. To disable the property in XML, for example:

```
<domain_participant_qos>
  <property>
    <value>
      <element>
        <name>dds.transport.UDPv4.builtin.disable_interface_tracking</name>
        <value>true</value>
      </element>
    </value>
  </property>
</domain_participant_qos>
```

17.2 Detection of Unreachable Locators

It is possible for a DomainParticipant to announce locators for endpoints that are temporarily or permanently unreachable from a different DomainParticipant.

For example, DomainParticipant 'A' may send to a different DomainParticipant 'B' one locator where the IP address corresponds to a subnet that is not reachable from DomainParticipant 'B'. In such case, the DomainParticipant 'B' running in a different subnet should not use this address to send information to the endpoints of DomainParticipant 'A'.

In Connext DDS 5.2.3 and earlier, the middleware did not have the ability to detect unreachable locators. This had two main consequences:

1. The middleware could waste CPU cycles and bandwidth sending messages to unreachable locators.
2. If the unreachable locator was a multicast locator, the destination endpoint would never receive live samples from the sender's endpoints. For best-effort communication, this would have resulted in never receiving samples. For reliable communication, this would have resulted in sending samples as repair traffic.

Connext DDS 5.3.0 introduces a new locator REACHABILITY PING mechanism, which the middleware can use to detect when an endpoint is not reachable at a locator; then it can stop using the locator to send data to the endpoint. For temporary disconnections, the middleware will be able to detect and use an endpoint’s locator that becomes reachable again. While data is not being sent to an unreachable locator, the middleware still sends periodic REACHABILITY PING messages to see if it is still unreachable.

The configuration of the REACHABILITY mechanism is done using the following DomainParticipant's QosPolicy values:
- participant_qos.discovery_config.locator_reachability_assert_period
- participant_qos.discovery_config.reachability_lease_duration
- participant_qos.discovery_config.locator_reachability_change_detection_period

For more information on these QoS values, see Table 9.11 DDS_DiscoveryConfigQosPolicy.

### 17.3 Using DNS Tracker to Keep Peer List Updated

Connext DDS allows the use of hostnames instead of IP addresses when configuring peers for specific transports (e.g., UDPv4 and UDPv6). By default, Connext DDS resolves hostnames into IP addresses only when the DomainParticipant is created. But you can use the DNS tracker to keep the IP addresses of these hostnames updated. The DNS tracker does this by creating a thread that regularly polls the DNS service. This thread detects changes in the IP address that a hostname is resolved to and updates the related peers accordingly.

Use the dns_tracker_polling_period field in the 9.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) on page 644 to define how often the DNS tracker thread will query the DNS service for updates (e.g., every 30 seconds). When the period is set to DDS_DURATION_INFINITE (the default value), the tracker is disabled and changes in hostnames will not be tracked. You can also configure the polling period after the creation of the DomainParticipant using the DomainParticipant’s set_dns_tracker_polling_period() operation. This operation can enable or disable the DNS tracker depending on the value of the DDS_Duration_t provided as parameter.

Connext DDS keeps information regarding the hostnames of peers, whether the hostnames are part of the initial_peers field in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640 or the peers were added through the DomainParticipant’s add_peer() API. When the DNS tracker is enabled, it builds the list of names to track from the DomainParticipant’s current peers. Removing peers from the DomainParticipant (using the remove_peer() API) will also affect the DNS tracker. If all the peers with a specific name are removed, the DNS tracker will stop tracking that name.

Enabling the DNS tracker changes the behavior of the DomainParticipant’s add_peer() API. If the DNS tracker has not been enabled, the API will fail to add a peer with a hostname that cannot be resolved into an IP address. If the DNS tracker has been enabled, the DomainParticipant’s add_peer() API will successfully add a peer with a hostname that cannot be resolved into an IP address. Although the hostname or the DNS service may not be available when the add_peer() API is called, the enabled DNS tracker will be able to resolve the name and update the locator once the hostname can be resolved.

Once the DNS tracker has been enabled, the change in the behavior of the DomainParticipant’s add_peer() API remains until the DomainParticipant is destroyed. Adding a peer with a hostname that cannot be resolved will not produce an error even if the DNS tracker is disabled. Connext DDS assumes that if the DNS tracker has been enabled once, all hostnames should be considered as valid peers independently of the status of the DNS tracker at the moment of adding the peer.
Chapter 18 Built-In Topics

This chapter discusses how to use Built-in Topics.

*Connext DDS* must discover and keep track of remote entities, such as new participants in the DDS domain. This information may also be important to the application itself, which may want to react to this discovery or access it on demand. To support these needs, *Connext DDS* provides *built-in Topics* (“DCPSParticipant”, “DCPSPublication”, “DCPSSubscription” in Figure 15.2: Built-in Writers and Readers for Discovery on page 773) and the corresponding built-in *DataReader* s that you can use to access this discovery information.

The discovery information is accessed just as if it is normal application data. This allows the application to know (either via listeners or by polling) when there are any changes in those values. Note that only entities that belong to a different *DomainParticipant* are being discovered and can be accessed through the built-in readers. Entities that are created within the local *DomainParticipant* are not included as part of the data that can be accessed by the built-in readers.

Built-in topics contain information about the remote entities, including their QoS policies. These QoS policies appear as normal fields inside the topic’s data, which can be read by means of the built-in Topic. Additional information is provided to identify the entity and facilitate the application logic.

18.1 Listeners for Built-in Entities

Built-in entities have default listener settings:

- The built-in *Subscriber* and its built-in topics have 'nil' listeners—all status bits are set in the listener masks, but the listener is NULL. This effectively creates a NO-OP listener that does not reset communication status.
- Built-in *DataReaders* have null listeners with no status bits set in their masks.

This approach prevents callbacks to the built-in *DataReader* listeners from invoking your *DomainParticipant’s* listeners, and at the same time ensures that the status changed flag is not
reset. For more information, see Table 4.4 Effect of Different Combinations of Listeners and Status Bit Masks and 4.4.5 Hierarchical Processing of Listeners on page 194.

18.2 Built-in DataReaders

Built-in DataReaders belong to a built-in Subscriber, which can be retrieved by using the DomainParticipant’s get_builtin_subscriber() operation. You can retrieve the built-in DataReaders by using the Subscriber’s lookup_datareader() operation, which takes the Topic name as a parameter. The built-in DataReader is created when lookup_datareader() is called on a built-in topic for the first time.

To conserve memory, built-in Subscribers and DataReaders are created only if and when you look them up. Therefore, if you do not want to miss any built-in data, you should look up the built-in readers before the DomainParticipant is enabled.

The following tables describe the built-in topics and their data types. The 7.5.30 USER_DATA QosPolicy on page 465, 5.2.1 TOPIC_DATA QosPolicy on page 224 and 7.4.4 GROUP_DATA QosPolicy on page 353 are included as part of the built-in data type and are not used by Connext DDS. Therefore, you can use them to send application-specific information.

Built-in topics can be used in conjunction with the ignore_*() operations to ignore certain entities (see 18.4 Restricting Communication—Ignoring Entities on page 841).

### Table 18.1 Participant Built-in Topic’s Data Type (DDS_ParticipantBuiltinTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_BuiltinTopicKey</td>
<td>key</td>
<td>Key to distinguish the discovered DomainParticipant</td>
</tr>
<tr>
<td>DDS_User-DataQosPolicy</td>
<td>user_data</td>
<td>Data that can be set when the related DomainParticipant is created (via the 7.5.30 USER_DATA QosPolicy on page 465) and the application may use as it wishes (e.g., to perform some security checking).</td>
</tr>
<tr>
<td>DDS_PropertyQosPolicy</td>
<td>property</td>
<td>Pairs of names/values to be stored with the DomainParticipant. See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439. The usage is strictly application-dependent.</td>
</tr>
<tr>
<td>DDS_ProtoctoolVersion_t</td>
<td>rtps_protoctool_version</td>
<td>Version number of the RTPS wire protocol used.</td>
</tr>
<tr>
<td>DDS_VendorId_t</td>
<td>rtps_vendor_id</td>
<td>ID of vendor implementing the RTPS wire protocol.</td>
</tr>
<tr>
<td>DDSUnsignedLong</td>
<td>dds_builtin_endpoints</td>
<td>Bitmap set by the discovery plugins. Each bit in this field indicates a built-in endpoint present for discovery.</td>
</tr>
<tr>
<td>DDS_LocaterSeq</td>
<td>defaultunicast_locators</td>
<td>If the TransportUnicastQosPolicy is not specified when a DataWriter/DataReader is created, the unicast locations in the corresponding Publication/Subscription built-in topic data will be empty. When the unicast_locations in the Publication/Subscription BuiltInTopicData is empty, the default_unicast_locations in the corresponding Participant Builtin Topic Data is assumed. If default_unicast_locations is empty, it defaults to DomainParticipantQos.default_unicast.</td>
</tr>
</tbody>
</table>
### Table 18.1 Participant Built-in Topic’s Data Type (DDS_ParticipantBuiltinTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_ProductVersion_t</td>
<td>product_version</td>
<td>Vendor-specific parameter. The current version of Connext DDS.</td>
</tr>
<tr>
<td>DDS_EntityNameQosPolicy</td>
<td>participant_name</td>
<td>Name and role_name assigned to the DomainParticipant. See 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
<tr>
<td>DDS_DomainId_t</td>
<td>domain_id</td>
<td>Domain ID associated with the discovered participant.</td>
</tr>
<tr>
<td>DDS_TransportInfoSeq</td>
<td>transport_info</td>
<td>A sequence of DDS_TransportInfo_t containing information about each of the installed transports of the discovered DomainParticipant. A DDS_TransportInfo_t structure contains the class_id and message_size_max for a single transport. The maximum length of this sequence is controlled by the 9.5.4 DOMAIN_PARTICIPANT_RESOURCELIMITS QosPolicy (DDS Extension) on page 654 transport_info_list_max_length (see Table 9.13 DDS_DomainParticipantResourceLimitsQosPolicy). Connext DDS uses the transport information propagated via discovery to detect potential misconfigurations in a Connext DDS distributed system. If two DomainParticipants that discover each other have one common transport with different values for message_size_max, Connext DDS prints a warning message about that condition.</td>
</tr>
<tr>
<td>DDS_ServiceQosPolicy</td>
<td>service</td>
<td>Service associated with the discovered DomainParticipant.</td>
</tr>
</tbody>
</table>

### Table 18.2 Publication Built-in Topic’s Data Type (DDS_PublicationBuiltinTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_BuiltinTopicKey_t</td>
<td>key</td>
<td>Key to distinguish the discovered DataWriter</td>
</tr>
<tr>
<td>DDS_BuiltinTopicKey_t</td>
<td>participant_key</td>
<td>Key to distinguish the participant to which the discovered DataWriter belongs</td>
</tr>
<tr>
<td>DDS_String</td>
<td>topic_name</td>
<td>Topic name of the discovered DataWriter</td>
</tr>
<tr>
<td>DDS_String</td>
<td>type_name</td>
<td>Type name attached to the topic of the discovered DataWriter</td>
</tr>
</tbody>
</table>
## Table 18.2 Publication Built-in Topic’s Data Type (DDS_PublicationBuiltInTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_DurabilityQosPolicy</td>
<td>durability</td>
<td>QosPolicies of the discovered DataWriter</td>
</tr>
<tr>
<td>DDS_DurabilityServiceQosPolicy</td>
<td>durability_service</td>
<td></td>
</tr>
<tr>
<td>DDS_DeadlineQosPolicy</td>
<td>deadline</td>
<td></td>
</tr>
<tr>
<td>DDS_DestinationOrderQosPolicy</td>
<td>destination_order</td>
<td></td>
</tr>
<tr>
<td>DDS_LatencyBudgetQosPolicy</td>
<td>latency_budget</td>
<td></td>
</tr>
<tr>
<td>DDS_LivelinessQosPolicy</td>
<td>liveliness</td>
<td></td>
</tr>
<tr>
<td>DDS_ReliabilityQosPolicy</td>
<td>reliability</td>
<td></td>
</tr>
<tr>
<td>DDS_LifespanQosPolicy</td>
<td>lifespan</td>
<td></td>
</tr>
<tr>
<td>DDS_UserDataQosPolicy</td>
<td>user_data</td>
<td>Data that can be set when the DataWriter is created (via the 7.5.30 USER_DATA QosPolicy on page 465) and that the application may use as it wishes.</td>
</tr>
<tr>
<td>DDS_OwnershipQosPolicy</td>
<td>ownership</td>
<td></td>
</tr>
<tr>
<td>DDS_OwnershipStrengthQosPolicy</td>
<td>ownership_strength</td>
<td>QosPolicies of the discovered DataWriter</td>
</tr>
<tr>
<td>DDS_DestinationOrderQosPolicy</td>
<td>destination_order</td>
<td></td>
</tr>
<tr>
<td>DDS_PresentationQosPolicy</td>
<td>presentation</td>
<td></td>
</tr>
<tr>
<td>DDS_PartitionQosPolicy</td>
<td>partition</td>
<td>Name of the partition, set in the 7.4.5 PARTITION QosPolicy on page 356 for the publisher to which the discovered DataWriter belongs</td>
</tr>
<tr>
<td>DDS_TopicDataQosPolicy</td>
<td>topic_data</td>
<td>Data that can be set when the Topic(with which the discovered DataWriter is associated) is created (via the 5.2.1 TOPIC_DATA QosPolicy on page 224) and that the application may use as it wishes.</td>
</tr>
<tr>
<td>DDS_GroupDataQosPolicy</td>
<td>group_data</td>
<td>Data that can be set when the Publisher to which the discovered DataWriter belongs is created (via the 7.4.4 GROUP_DATA QosPolicy on page 353) and that the application may use as it wishes.</td>
</tr>
<tr>
<td>DDS_TypeObject*</td>
<td>type</td>
<td>Describes the type of the remote DataReader. See the API Reference HTML documentation.</td>
</tr>
<tr>
<td>DDS_DataRepresentationQosPolicy</td>
<td>representation</td>
<td>Data representations that the DataWriter offers. See 7.5.3 DATA_REPRESENTATION QosPolicy on page 380.</td>
</tr>
<tr>
<td>DDS_DataTagQosPolicy</td>
<td>data_tags</td>
<td>Data tags (pairs of names/values) assigned to the corresponding DataWriter. Usage is strictly application-dependent. See 7.5.4 DATATAG QosPolicy on page 388.</td>
</tr>
</tbody>
</table>
18.2 Built-in DataReaders

Table 18.2 Publication Built-in Topic’s Data Type (DDS_PublisherBuiltInTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TypeCode*</td>
<td>type_code</td>
<td>Type code information about this Topic. See 3.7 Using Generated Types without Connext DDS (Standalone) on page 152.</td>
</tr>
<tr>
<td>DDS_BuiltinTopicKey_t</td>
<td>publisher_key</td>
<td>The key of the Publisher to which the DataWriter belongs.</td>
</tr>
<tr>
<td>DDS_PropertyQosPolicy</td>
<td>property</td>
<td>Properties (pairs of names/values) assigned to the corresponding DataWriter. Usage is strictly application-dependent. See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.</td>
</tr>
<tr>
<td>DDS_LocatorSeq</td>
<td>unicast_locators</td>
<td>If the TransportUnicastQosPolicy is not specified when a DataWriter/DataReader is created, the unicast_locators in the corresponding Publication/Subscription built-in topic data will be empty. When the unicast_locators in the Publication/SubscriptionBuiltinTopicData is empty, the default_unicast_locators in the corresponding Participant Built-in Topic Data is assumed.</td>
</tr>
<tr>
<td>DDS_GUID_t</td>
<td>virtual_guid</td>
<td>Virtual GUID for the corresponding DataWriter. For more information, see 13.2 Durability and Persistence Based on Virtual GUIDs on page 737.</td>
</tr>
<tr>
<td>DDS_ServiceQosPolicy</td>
<td>service</td>
<td>Service associated with the discovered DataWriter.</td>
</tr>
<tr>
<td>DDS_ProtocolVersion_t</td>
<td>rtps_protocol_version</td>
<td>Version number of the RTPS wire protocol in use.</td>
</tr>
<tr>
<td>DDS_VendorId_t</td>
<td>rtps_vendor_id</td>
<td>ID of the vendor implementing the RTPS wire protocol.</td>
</tr>
<tr>
<td>DDS_Product_Version_t</td>
<td>product_version</td>
<td>Vendor-specific value. For RTI, this is the current version of Connext DDS.</td>
</tr>
<tr>
<td>DDS_LocatorFilterQosPolicy</td>
<td>locator_filter</td>
<td>When the 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432 is used on the discovered DataWriter, the locator_filter contains the sequence of LocatorFilters in that policy. There is one LocatorFilter per DataWriter channel. A channel is defined by a filter expression and a sequence of multicast locators. See 18.2.1 LOCATOR_FILTER Qos Policy (DDS Extension) on page 840.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>disable_positive_acks</td>
<td>Vendor specific parameter. Determines whether matching DataReaders send positive acknowledgements for reliability.</td>
</tr>
<tr>
<td>DDS_EntityNameQosPolicy</td>
<td>publication_name</td>
<td>Name and role_name assigned to the DataWriter. See 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
</tbody>
</table>

Table 18.3 Subscription Built-in Topic’s Data Type (DDS_SubscriptionBuiltInTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_BuiltinTopicKey_t</td>
<td>key</td>
<td>Key to distinguish the discovered DataReader.</td>
</tr>
<tr>
<td>DDS_BuiltinTopicKey_t</td>
<td>participant_key</td>
<td>Key to distinguish the participant to which the discovered DataReader belongs.</td>
</tr>
</tbody>
</table>
### Table 18.3 Subscription Built-in Topic’s Data Type (DDS_Subscription builtinTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>char*</td>
<td>topic_name</td>
<td>Topic name of the discovered <em>DataReader</em>.</td>
</tr>
<tr>
<td>char*</td>
<td>type_name</td>
<td>Type name attached to the <em>Topic</em> of the discovered <em>DataReader</em>.</td>
</tr>
<tr>
<td>DDS_DurabilityQosPolicy</td>
<td>durability</td>
<td>QoS Policies of the discovered <em>DataReader</em>.</td>
</tr>
<tr>
<td>DDS_DeadlineQosPolicy</td>
<td>deadline</td>
<td></td>
</tr>
<tr>
<td>DDS_LatencyBudgetQosPolicy</td>
<td>latency_budget</td>
<td></td>
</tr>
<tr>
<td>DDS_LivelinessQosPolicy</td>
<td>liveliness</td>
<td></td>
</tr>
<tr>
<td>DDS_ReliabilityQosPolicy</td>
<td>reliability</td>
<td></td>
</tr>
<tr>
<td>DDS_OwnershipQosPolicy</td>
<td>ownership</td>
<td></td>
</tr>
<tr>
<td>DDS_DestinationOrderQosPolicy</td>
<td>destination_order</td>
<td>Data that can be set when the <em>DataReader</em> is created (via the 7.5.30 USER_DATA QosPolicy on page 465) and that the application may use as it wishes.</td>
</tr>
<tr>
<td>DDS_UserDataQosPolicy</td>
<td>user_data</td>
<td></td>
</tr>
<tr>
<td>DDS_TimeBasedFilterQosPolicy</td>
<td>time_based_filter</td>
<td>QoS Policies of the discovered <em>DataReader</em>.</td>
</tr>
<tr>
<td>DDS_PresentationQosPolicy</td>
<td>presentation</td>
<td></td>
</tr>
<tr>
<td>DDS_PartitionQosPolicy</td>
<td>partition</td>
<td>Name of the partition, set in the 7.4.5 PARTITION QosPolicy on page 356 for the <em>Subscriber</em> to which the discovered <em>DataReader</em> belongs.</td>
</tr>
<tr>
<td>DDS_TopicDataQosPolicy</td>
<td>topic_data</td>
<td>Data that can be set when the <em>Topic</em> to which the discovered <em>DataReader</em> belongs is created (via the 5.2.1 TOPIC_DATA QosPolicy on page 224) and that the application may use as it wishes.</td>
</tr>
<tr>
<td>DDS_GroupDataQosPolicy</td>
<td>group_data</td>
<td>Data that can be set when the <em>Publisher</em> to which the discovered <em>DataReader</em> belongs is created (via the 7.4.4 GROUP_DATA QosPolicy on page 353) and that the application may use as it wishes.</td>
</tr>
<tr>
<td>DDS_TypeObject *</td>
<td>type</td>
<td>Describes the type of the remote <em>DataReader</em>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>See the API Reference HTML documentation.</td>
</tr>
<tr>
<td>DDS_TypeConsistencyEnforcementQosPolicy</td>
<td>type_consistency</td>
<td>Indicates the type-consistency requirements of the remote <em>DataReader</em>. See 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594 and the RTI Connect DDS Core Libraries Extensible Types Guide.</td>
</tr>
<tr>
<td>DDS_DataRepresentationQosPolicy</td>
<td>representation</td>
<td>Data representations that the <em>DataReader</em> requests. See 7.5.3 DATA_REPRESENTATION QosPolicy on page 380.</td>
</tr>
<tr>
<td>DDS_DataTagQosPolicy</td>
<td>data_tags</td>
<td>Data tags (pairs of names/values) assigned to the corresponding <em>DataReader</em>. Usage is strictly application-dependent. See 7.5.4 DATATAG QosPolicy on page 388.</td>
</tr>
</tbody>
</table>
Table 18.3 Subscription Built-in Topic’s Data Type (DDS_SubscriptionBuiltinTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_TypeCode *</td>
<td>type_code</td>
<td>Type code information about this Topic. See 3.7 Using Generated Types without Con-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Next DDS (Standalone) on page 152.</td>
</tr>
<tr>
<td>DDS_BuiltinTopicKey_t</td>
<td>subscriber_key</td>
<td>Key of the Subscriber to which the DataReader belongs.</td>
</tr>
<tr>
<td>DDS_PropertyQosPolicy</td>
<td>property</td>
<td>Properties (pairs of names/values) assigned to the corresponding DataReader. Usage is strictly application-dependent. See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.</td>
</tr>
<tr>
<td>DDS_LocatorSeq</td>
<td>unicast_locators</td>
<td>If the TransportUnicastQosPolicy is not specified when a DataWriter/DataReader is created, the unicast_locators in the corresponding Publication/Subscription builtin topic data will be empty. When the unicast_locators in the Publication/SubscriptionBuiltinTopicData is empty, the default_unicast_locators in the corresponding Participant Builtin Topic Data is assumed.</td>
</tr>
<tr>
<td>DDS_LocatorSeq</td>
<td>multicast_locators</td>
<td>Custom multicast locators that the endpoint can specify.</td>
</tr>
<tr>
<td>DDS_ContentFilter-Property_t</td>
<td>content_filter_property</td>
<td>Provides all the required information to enable content filtering on the writer side.</td>
</tr>
<tr>
<td>DDS_GUID_t</td>
<td>virtual_guid</td>
<td>Virtual GUID for the corresponding DataReader. For more information, see 13.2 Durability and Persistence Based on Virtual GUIDs on page 737.</td>
</tr>
<tr>
<td>DDS_ServiceQosPolicy</td>
<td>service</td>
<td>Service associated with the discovered DataReader.</td>
</tr>
<tr>
<td>DDS_ProtocolVersion_t</td>
<td>rtps_protocol_version</td>
<td>Version number of the RTPS wire protocol in use.</td>
</tr>
<tr>
<td>DDS_VendorId_t</td>
<td>rtps_vendor_id</td>
<td>ID of the vendor implementing the RTPS wire protocol.</td>
</tr>
<tr>
<td>DDS_Product_Version_t</td>
<td>product_version</td>
<td>Vendor-specific value. For RTI, this is the current version of Connext DDS.</td>
</tr>
<tr>
<td>DDS_Boolean</td>
<td>disable_positive_acks</td>
<td>Vendor specific parameter. Determines whether matching DataReaders send positive acknowledgements for reliability.</td>
</tr>
<tr>
<td>DDS_EntityNameQosPolicy</td>
<td>subscription_name</td>
<td>Name and role_name assigned to the DataReader. See 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418.</td>
</tr>
</tbody>
</table>

Table 18.4 Topic Built-in Topic’s Data Type (DDS_TopicBuiltinTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_BuiltinTopicKey_t</td>
<td>key</td>
<td>Key to distinguish the discovered Topic</td>
</tr>
</tbody>
</table>
Table 18.4 Topic Built-in Topic’s Data Type (DDS_TopicBuiltinTopicData)

<table>
<thead>
<tr>
<th>Type</th>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_String</td>
<td>name</td>
<td>Topic name</td>
</tr>
<tr>
<td>DDS_String</td>
<td>type_name</td>
<td>type name attached to the Topic</td>
</tr>
<tr>
<td>DDS_DurabilityQosPolicy</td>
<td>durability</td>
<td></td>
</tr>
<tr>
<td>DDS_DurabilityServiceQosPolicy</td>
<td>durability_service</td>
<td></td>
</tr>
<tr>
<td>DDS_DeadlineQosPolicy</td>
<td>deadline</td>
<td></td>
</tr>
<tr>
<td>DDS_LatencyBudgetQosPolicy</td>
<td>latency_budget</td>
<td></td>
</tr>
<tr>
<td>DDS_LivelinessQosPolicy</td>
<td>liveliness</td>
<td></td>
</tr>
<tr>
<td>DDS_ReliabilityQosPolicy</td>
<td>reliability</td>
<td></td>
</tr>
<tr>
<td>DDS_TransportPriorityQosPolicy</td>
<td>transport_priority</td>
<td>QosPolicy of the discovered Topic</td>
</tr>
<tr>
<td>DDS_LifespanQosPolicy</td>
<td>lifespan</td>
<td></td>
</tr>
<tr>
<td>DDS_DestinationOrderQosPolicy</td>
<td>destination_order</td>
<td></td>
</tr>
<tr>
<td>DDS_HistoryQosPolicy</td>
<td>history</td>
<td></td>
</tr>
<tr>
<td>DDS_ResourceLimitsQosPolicy</td>
<td>resource_limits</td>
<td></td>
</tr>
<tr>
<td>DDS_OwnershipQosPolicy</td>
<td>ownership</td>
<td></td>
</tr>
<tr>
<td>DDS_TopicDataQosPolicy</td>
<td>topic_data</td>
<td>Data that can be set when the Topic to which the discovered DataReader belongs is created (via the 5.2.1 TOPIC_DATA QosPolicy on page 224) and that the application may use as it wishes.</td>
</tr>
</tbody>
</table>

Table 18.5 QoS of Built-in Subscriber and DataReader lists the QoS of the built-in Subscriber and DataReader created for accessing discovery data. These are provided for your reference only; they cannot be changed.

Table 18.5 QoS of Built-in Subscriber and DataReader

<table>
<thead>
<tr>
<th>QosPolicy</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deadline</td>
<td>period = infinite</td>
</tr>
<tr>
<td>DestinationOrder</td>
<td>kind = BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</td>
</tr>
<tr>
<td>Durability</td>
<td>kind = TRANSIENT_LOCAL_DURABILITY_QOS</td>
</tr>
</tbody>
</table>
### Table 18.5 QoS of Built-in Subscriber and DataReader

<table>
<thead>
<tr>
<th>QoSPolicy</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EntityFactory</td>
<td>autoenable_created_entities = TRUE</td>
</tr>
<tr>
<td>GroupData</td>
<td>value = empty sequence</td>
</tr>
<tr>
<td>History</td>
<td>kind = KEEP_LAST_HISTORY_QOS</td>
</tr>
<tr>
<td></td>
<td>depth = 1</td>
</tr>
<tr>
<td>LatencyBudget</td>
<td>duration = 0</td>
</tr>
<tr>
<td>Liveliness</td>
<td>kind = AUTOMATIC_LIVENESS_QOS</td>
</tr>
<tr>
<td></td>
<td>lease_duration = infinite</td>
</tr>
<tr>
<td>Ownership</td>
<td>kind = SHARED.OwnerSHIP_QOS</td>
</tr>
<tr>
<td>Ownership Strength</td>
<td>value = 0</td>
</tr>
<tr>
<td>Presentation</td>
<td>access_scope = TOPIC_PRESENTATION_QOS</td>
</tr>
<tr>
<td></td>
<td>coherent_access = FALSE</td>
</tr>
<tr>
<td></td>
<td>ordered_access = FALSE</td>
</tr>
<tr>
<td>Partition</td>
<td>name = empty sequence</td>
</tr>
<tr>
<td>ReaderDataLifecycle</td>
<td>autopurge_nowriter_samples_delay = infinite</td>
</tr>
<tr>
<td>Reliability</td>
<td>kind = RELIABLE.RELIABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>max_blocking_time is irrelevant for the DataReader</td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>Depends on setting of DomainParticipantResourceLimitsQosPolicy and DiscoveryConfigQosPolicy in DomainParticipantQos:</td>
</tr>
<tr>
<td></td>
<td>max_samples = domainParticipantQos.discovery_config.</td>
</tr>
<tr>
<td></td>
<td>[participant/publication/subscription].reader_resource_limits.max_samples</td>
</tr>
<tr>
<td></td>
<td>max_instances = domainParticipantQos.resource_limits.</td>
</tr>
<tr>
<td></td>
<td>[remote_writer/reader/participant].allocation.max_count</td>
</tr>
<tr>
<td></td>
<td>max_samples_per_instance = 1</td>
</tr>
<tr>
<td>TimeBasedFilter</td>
<td>minimum_separation = 0</td>
</tr>
<tr>
<td>TopicData</td>
<td>value = empty sequence</td>
</tr>
<tr>
<td>UserData</td>
<td>value = empty sequence</td>
</tr>
</tbody>
</table>

**Note:**

The DDS_TopicBuiltinTopicData built-in topic (described in Table 18.4 Topic Built-in Topic’s Data Type (DDS_TopicBuiltinTopicData)) is meant to convey information about discovered Topics. However, this topic's data is not sent separately and therefore a DataReader for DDS_TopicBuiltinTopicData will not receive any data. Instead, DDS_TopicBuiltinTopicData data is included in the information carried by the built-in topics for Publications and Subscriptions (DDS_PublicationBuiltinTopicData and DDS_SubscriptionBuiltinTopicData) and can be accessed with their built-in DataReaders.
18.2.1 LOCATOR_FILTER QoS Policy (DDS Extension)

The LocatorFilter QoS Policy is only applicable to the built-in topic for a Publication (see Table 18.2 Publication Built-in Topic’s Data Type (DDS_PublicationBuiltInTopicData)).

Table 18.6 DDS_LocatorFilterQosPolicy

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LocatorFilterSeq</td>
<td>locator_filters</td>
<td>A sequence of locator filters, described in Table 18.7 DDS_LocatorFilter_t. There is one locator filter per DataWriter channel. If the length of the sequence is zero, the DataWriter is not using multi-channel.</td>
</tr>
<tr>
<td>char*</td>
<td>filter_name</td>
<td>Name of the filter class used to describe the locator filter expressions. The following two values are supported:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_SQLFILTER_NAME</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• DDS_STRINGMATCHFILTER_NAME</td>
</tr>
</tbody>
</table>

Table 18.7 DDS_LocatorFilter_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_LocatorSeq</td>
<td>locators</td>
<td>A sequence of multicast address locators for the locator filter. See Table 18.8 DDS_Locator_t.</td>
</tr>
<tr>
<td>char*</td>
<td>filter_expression</td>
<td>A logical expression used to determine if the data will be published in the channel associated with this locator filter. See 5.4.6 SQL Filter Expression Notation on page 236 and 5.4.7 STRINGMATCH Filter Expression Notation on page 245 for information about the expression syntax.</td>
</tr>
</tbody>
</table>

Table 18.8 DDS_Locator_t

<table>
<thead>
<tr>
<th>Type</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long(kind)</td>
<td>kind</td>
<td>If the locator kind is DDS_LOCATOR_KIND_UDPv4, the address contains an IPv4 address. The leading 12 octets of the address must be zero. The last 4 octets store the IPv4 address. If the locator kind is DDS_LOCATOR_KIND_UDPv6, the address contains an IPv6 address. IPv6 addresses typically use a shorthand hexadecimal notation that maps one-to-one to the 16 octets of the address. In C#, the locator kinds for UDPv4 and UDPv6 addresses are Locator_t.LOCATOR_KIND_UDPv4 and Locator_t.LOCATOR_KIND_UDPv6.</td>
</tr>
<tr>
<td>DDS_Octet<a href="address">16</a></td>
<td>address</td>
<td>The locator address.</td>
</tr>
<tr>
<td>DDS_UnsignedLong(port)</td>
<td>port</td>
<td>The locator port number.</td>
</tr>
</tbody>
</table>
18.3 Accessing the Built-in Subscriber

Getting the built-in subscriber allows you to retrieve the built-in readers of the built-in topics through the Subscriber’s `lookup_datareader()` operation. By accessing the built-in reader, you can access discovery information about remote entities.

```cpp
// Lookup built-in reader
DDSDataReader *builtin_reader =
  builtin_subscriber->lookup_datareader(DDS/Publication/Topic/Name);
if (builtin_reader == NULL) {
  // ... error
}
// Register listener to built-in reader
MyPublicationBuiltInTopicDataListener builtin_reader_listener =
  new MyPublicationBuiltInTopicDataListener();
if (builtin_reader->set_listener(builtin_reader_listener,
    DDS/Available/Status) != DDS/Retcode/Ok) {
  // ... error
}
// enable DomainParticipant
if (participant->enable() != DDS/Retcode/Ok) {
  // ... error
}
```

For example, you can call the DomainParticipant’s `get_builtin_subscriber()` operation, which will provide you with a built-in Subscriber. Then you can use that built-in Subscriber to call the Subscriber’s `lookup_datareader()` operation; this will retrieve the built-in reader. Another option is to register a Listener on the built-in subscriber instead, or poll for the status of the built-in subscriber to see if any of the built-in data readers have received data.

18.4 Restricting Communication—Ignoring Entities

The `ignore_participant()` operation allows an application to ignore all communication from a specific DomainParticipant. Or for even finer control you can use the `ignore_publication()`, `ignore_subscription()`, and `ignore_topic()` operations. These operations are described below.

```cpp
DDS/Return/Code_t ignore_participant (const DDS/Instance/Handle_t &handle)
DDS/Return/Code_t ignore_publication (const DDS/Instance/Handle_t &handle)
DDS/Return/Code_t ignore_subscription (const DDS/Instance/Handle_t &handle)
DDS/Return/Code_t ignore_topic (const DDS/Instance/Handle_t &handle)
```

The entity to ignore is identified by the `handle` argument. It may be a local or remote entity. For `ignore_publication()`, the handle will be that of a local DataWriter or a discovered remote DataWriter. For `ignore_subscription()`, that handle will be that of a local DataReader or a discovered remote DataReader.

The safest approach for ignoring an entity is to call the ignore operation within the Listener callback of the built-in reader, or before any local entities are enabled. This will guarantee that the local entities (entities that are created by the local DomainParticipant) will never have a chance to establish communication with the remote entities (entities that are created by another DomainParticipant) that are going to be ignored.
If the above is not possible and a remote entity is to be ignored after the communication channel has been established, the remote entity will still be removed from the database of the local application as if it never existed. However, since the remote application is not aware that the entity is being ignored, it may potentially be expecting to receive messages or continuing to send messages. Depending on the QoS of the remote entity, this may affect the behavior of the remote application and may potentially stop the remote application from communicating with other entities.

You can use this operation in conjunction with the ParticipantBuiltInTopicData to implement access control. You can pass application data associated with a DomainParticipant in the 7.5.30 USER_DATA QosPolicy on page 465. This application data is propagated as a field in the built-in topic. Your application can use the data to implement an access control policy.

Ignore operations, in conjunction with the Built-in Topic Data, can be used to implement access control. You can pass data associated with an entity in the 7.5.30 USER_DATA QosPolicy on page 465, 7.4.4 GROUP_DATA QosPolicy on page 353 or 5.2.1 TOPIC_DATA QosPolicy on page 224. This data is propagated as a field in the built-in topic. When data for a built-in topic is received, the application can check the user_data, group_data or topic_data field of the remote entity, determine if it meets the security requirement, and ignore the remote entity if necessary.

See also: Discovery (Chapter 15 on page 764).

18.4.1 Ignoring Specific Remote DomainParticipants

The ignore_participant() operation is used to instruct Connext DDS to locally ignore a remote DomainParticipant. It causes Connext DDS to locally behave as if the remote DomainParticipant does not exist.

```c
DDS_ReturnCode_t ignore_participant (const DDS_InstanceHandle_t & handle)
```

After invoking this operation, Connext DDS will locally ignore any Topic, publication, or subscription that originates on that DomainParticipant. (If you only want to ignore specific publications or subscriptions, see 18.4.2 Ignoring Publications and Subscriptions on the next page instead.) Figure 18.1: Ignoring Participants below provides an example.

By default, the maximum number of participants that can be ignored is limited by ignored_entity_allocation_max_count in the 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654. However, that behavior can be changed by using ignore_entity_replacement_kind in the same QoS policy.

See also: 18.4.4 Resource Limits Considerations for Ignored Entities on page 845.

Caution: There is no way to reverse this operation. You can add to the peer list, however—see 9.5.2.3 Adding and Removing Peers List Entries on page 641.

Figure 18.1: Ignoring Participants

```c
class MyParticipantBuiltInTopicDataListener :
public DDSDataReaderListener {
    public:
```
virtual void on_data_available(DDSDataReader *reader);
    // .......
};
void MyParticipantBuiltinTopicdataListener::on_data_available(DDSDataReader *reader) {
    DDSParticipantBuiltinTopicDataDataReader
        *builtinTopicDataReader =
            DDSParticipantBuiltinTopicDataDataReader(reader);
    DDS_BuiltinTopicDataSeq data_seq;
    DDS_SampleInfoSeq info_seq;
    int i = 0;
    if (builtinTopicDataReader->take(data_seq, info_seq,
            DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
            DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE) !=
            DDS_RETCODE_OK) {
        // ... error
    }
    for (i = 0; i < data_seq.length(); ++i) {
        if (info_seq[i].valid_data) {
            // check user_data for access control
            if (data_seq[i].user_data[0] != 0x9) {
                if (builtinTopicDataReader->get_subscriber()
                    ->get_participant() ->ignore_participant( info_seq[i].instance_handle)
                    != DDS_RETCODE_OK) {
                    // ... error
                }
            };
        }
    }
    if (builtinTopicDataReader->return_loan( data_seq, info_seq) != DDS_RETCODE_OK) {
        // ... error
    }
}

18.4.2 Ignoring Publications and Subscriptions

You can instruct Connext DDS to locally ignore a publication or subscription. A publication/subscription is defined by the association of a Topic name, user data and partition set on the Publisher/Subscriber. After this call, any data written related to associated DataWriter/DataReader will be ignored.

The entity to ignore is identified by the handle argument. For ignore_publication(), the handle will be that of a DataWriter. For ignore_subscription(), that handle will be that of a DataReader.

This operation can be used to ignore local and remote entities:

- For local entities, you can obtain the handle argument by calling the get_instance_handle() operation for that particular entity.
- For remote entities, you can obtain the handle argument from the DDS_SampleInfo structure retrieved when reading DDS data samples available for the entity’s built-in DataReader.
18.4.3 Ignoring Topics

The `ignore_topic()` operation instructs Connext DDS to locally ignore a Topic. This means it will locally ignore any publication or subscription to the Topic.

```c
DDS_ReturnCode_t ignore_topic (const DDS_InstanceHandle_t & handle)
```

---

**Caution:** There is no way to reverse these operations.

Figure 18.2: Ignoring Publications below provides an example.

```c
class MyPublicationBuiltinTopicDataListener : public DDSDataReaderListener
{
    public:
        virtual void on_data_available(DDSDataReader *reader);
        // ....
};
void MyPublicationBuiltinTopicDataListener::on_data_available(DDSDataReader *reader) {
    DDSPublicationBuiltinTopicDataListener *builtinTopicDataReader =
        (DDS_PublicationBuiltinTopicDataListener *)reader;
    DDS_PublicationBuiltinTopicDataSeq data_seq;
    DDS_SampleInfoSeq info_seq;
    int i = 0;
    if (builtinTopicDataReader->take(data_seq, info_seq,
        DDS_LENGTH_UNLIMITED, DDS_ANY_SAMPLE_STATE,
        DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE)
        != DDS_RETCODE_OK)
    {
        // ... error
    }
    for (i = 0; i < data_seq.length(); ++i) {
        if (info_seq[i].valid_data) {
            // check user_data for access control
            if (data_seq[i].user_data[0] != 0x9) {
                if (builtinTopicDataReader->get_subscriber()
                    ->get_participant()
                    ->ignore_publication(
                        info_seq[i].instance_handle)
                    != DDS_RETCODE_OK) {
                    // ... error
                }
            }
        }
    }
    if (builtinTopicDataReader->return_loan(data_seq, info_seq) !=
        DDS_RETCODE_OK) {
        ...
    }
}
```
18.4.4 Resource Limits Considerations for Ignored Entities

Caution: There is no way to reverse this operation.

If you know that your application will never publish or subscribe to data under certain topics, you can use this operation to save local resources.

The Topic to ignore is identified by the handle argument. This handle is the one that appears in the DDS_SampleInfo retrieved when reading the DDS data samples from the built-in DataReader to the Topic.

18.4.4 Resource Limits Considerations for Ignored Entities

When an entity is ignored, Connext DDS adds it to an internal ‘ignore’ table whose resource limits are configured using the ignored_entity_allocation.max_count in the 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654. The behavior of Connext DDS when this limit is exceeded can be modified by using the ignored_entity_replacement_kind in the same QoS policy.

The default value for ignored_entity_replacement_kind is DDS_NO_REPLACEMENT_IGNORED_ENTITY_REPLACEMENT, meaning that a call to the DomainParticipant's ignore_participant(), ignore_publication(), or ignore_subscription() will fail if the DomainParticipant has ignored more entities than the limit set in ignored_entity_allocation.max_count entities.

When ignored_entity_replacement_kind is set to DDS_NOT_ALIVE_FIRST_IGNORED_ENTITY_REPLACEMENT, a call to ignore_participant() will not fail when ignored_entity_allocation.max_count is exceeded, as long as there is one DomainParticipant already ignored. Instead, the call will replace one of the existing DomainParticipants in the internal table. The remote DomainParticipant that will be replaced is the one for which the local DomainParticipant had not received any message for the longest time.

When a remote DomainParticipant is replaced in the ‘ignore’ table, it becomes un-ignored. Thus, the local DomainParticipant would have to call ignore_participant() again to re-ignore the replaced entity.

Note: In this release, ignored publications and subscriptions are never replaced in the ‘ignore’ table. Since this table also contains the ignored DomainParticipants, a call to ignore_participant() will fail if ignored_entity_allocation.max_count is reached and none of the ignored entities is a DomainParticipant.

18.4.5 Supervising Endpoint Discovery

It is possible to control for which DomainParticipants endpoint discovery may occur. You can configure this behavior with the enable_endpoint_discovery field in the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640:

- When set to TRUE (the default value), endpoint discovery will automatically occur for every discovered DomainParticipant. This is the normal operation of the discovery process.
• When set to FALSE, endpoint discovery will be disabled for every discovered DomainParticipant. Then applications will have to manually enable endpoint discovery (described below) for the DomainParticipants they are interested in communicating with. By disabling endpoint discovery, the DomainParticipant will not store any state about remote endpoints and will not send local endpoint information to remote DomainParticipants.

When enable_endpoint_discovery is set to FALSE, you have two options after a remote DomainParticipant is discovered:

• Call the DomainParticipant’s resume_endpoint_discovery() operation to enable endpoint discovery. After invoking this operation, the DomainParticipant will start to exchange endpoint information so that matching and communication can occur with the remote DomainParticipant.

```cpp
DDS_ReturnCode_t resume_endpoint_discovery(
    const DDS_InstanceHandle_t & remoteParticipant_handle)
```

Or

• Call the DomainParticipant’s ignore_participant() operation to permanently ignore endpoint discovery with the remote DomainParticipant.

Setting enable_endpoint_discovery to FALSE enables application-level authentication use cases, in which a DomainParticipant will resume endpoint discovery with a remote DomainParticipant after successful authentication at the application level. The following example shows how to provide access control using this feature:

```cpp
class MyParticipantBuiltInTopicDataListener : 
    public DDSDataReaderListener {
    public:
        virtual void on_data_available(DDSDataReader *reader); 
        // ...
    };

void MyParticipantBuiltInTopicDataListener::on_data_available( 
    DDSDataReader *reader) {
    DDSParticipantBuiltInTopicDataDataReader
        *builtinTopicDataReader = 
        DDSParticipantBuiltInTopicDataDataReader *(reader);
    DDS_ParticipantBuiltInTopicDataSeq data_seq;
    DDS_SampleInfoSeq info_seq;
    int i = 0;
    if (builtinTopicDataReader->take( 
            data_seq, info_seq, 
            DDS_LENGTH_UNLIMITED, 
            DDS_ANY_SAMPLE_STATE, 
            DDS_ANY_VIEW_STATE, 
            DDS_ANY_INSTANCE_STATE) != DDS_RETCODE_OK) {
            // ... error
        }
    for (i = 0; i < data_seq.length(); ++i) {
        if (info_seq[i].valid_data) {
```
DDSDomainParticipant * localParticipant =
builtinTopicDataReader->
get_subscriber()->get_participant();
DDS_ReturnCode_t retCode;
// check user data for access control
if (data_seq[i].user_data[0] != 0x9) {
    retCode = localParticipant->
        ignore_participant(
            info_seq[i].instance_handle);
} else {
    retCode = localParticipant->
        resume_endpoint_discovery(
            info_seq[i].instance_handle)
        
}
}
if (builtinTopicDataReader->return_loan(
    data_seq, info_seq)
    != DDS_RETCODE_OK) {
    // ... error
}
Chapter 19 Configuring QoS with XML

*Connext DDS* entities are configured by means of Quality of Service (QoS) policies, which may be set programmatically in one of the following ways:

- Directly when the entity is created as an additional argument to the `create_<entity>()` operation (or the Entity's constructor in the Modern C++ API).
- Directly via the `set_qos()` operation on the entity.
- Indirectly as a default QoS on the factory for the entity (`set_default_<entity>_qos()` operations on `Publisher`, `Subscriber`, `DomainParticipant`, `DomainParticipantFactory`)

Entities can also be configured from an XML file or XML string. With this feature, you can change QoS configurations simply by changing the XML file or string—you do not have to recompile the application. This chapter describes how to configure *Connext DDS* entities using XML.

### 19.1 Example XML File

The QoS configuration of a *Entity* can be loaded from an XML file or string.

The file contents must follow an important hierarchy: the file contains one or more libraries; each library contains one or more profiles; each profile contains QoS settings.

Let's look at a very basic configuration file, just to get an idea of its contents. You will learn the meaning of each line as you read the rest of this chapter:

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A XML configuration file -->
<dds version = 5.0.0>
  <qos_library name="RTILibrary">
    <!-- A QoS Profile is a set of related QoS -->
    <qos_profile name="StrictReliableCommunicationProfile">
      <datawriter_qos>
        <history>
          <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
      </datawriter_qos>
    </qos_profile>
  </qos_library>
</dds>
```
19.2 QoS Libraries

A QoS Library is a named set of QoS profiles.

One configuration file may have several QoS libraries, each one defining its own QoS profiles.

All QoS libraries must be declared within <dds> and </dds> tags. For example:

```xml
<dds>
  <qos_library name="RTILibrary">
    <!-- Individual QoS are shortcuts for QoS Profiles with 1 QoS -->
    <datawriter_qos name="KeepAllWriter">
      <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
      </history>
    </datawriter_qos>
  </qos_library>
  <qos_profile name="StrictReliableCommunicationProfile">
    <datawriter_qos>
      <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
      </history>
    </datawriter_qos>
  </qos_profile>
</dds>
```

See `<NDDSHOME>/resource/xml/NDDS_QOS_PROFILES.example.xml` for another example; this file contains the default QoS values for all entity kinds.
A QoS library can be reopened within the same configuration file or across different configuration files. For example:

```xml
<dds>
  <qos_library name="RTILibrary">
    ...
  </qos_library>
  ...
  <qos_library name="RTILibrary">
    ...
  </qos_library>
</dds>
```

19.3 QoS Profiles

A QoS Profile groups a set of related QoS policies, by entity (e.g., `<datawriter_qos>`), identified by a name. For example:

```xml
<qos_profile name="StrictReliableCommunicationProfile">
  <datawriter_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datawriter_qos>
  <datareader_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datareader_qos>
</qos_profile>
```

Duplicate QoS profiles are not allowed. To overwrite a QoS profile, use 19.3.3 QoS Profile Inheritance and Composition on page 854.

There are functions that allow you to create Entities using profiles, such as `create_participant_with_profile()` (9.3.1 Creating a DomainParticipant on page 615), `create_topic_with_profile()` (5.1.1 Creating Topics on page 217), etc.
If you create an entity using a profile without a QoS definition or an inherited QoS definition (see 19.3.3 QoS Profile Inheritance and Composition on page 854) for that class of entity, Connext DDS uses the default QoS.

Example 1:

```xml
<qos_profile name="BatchStrictReliableCommunicationProfile">
  <datawriter_qos>
    <batch>
      <enable>true</enable>
    </batch>
  </datawriter_qos>
</qos_profile>
```

The DataReader QoS value in the profile BatchStrictReliableCommunicationProfile is inherited from the profile StrictReliableCommunicationProfile.

Example 2:

```xml
<qos_profile name="BatchProfile">
  <datawriter_qos>
    <batch>
      <enable>true</enable>
    </batch>
  </datawriter_qos>
</qos_profile>
```

The DataReader QoS value in the profile BatchProfile is the default Connext DDS QoS.

19.3.1 Built-in QoS Profiles

Several QoS Profiles are built into the Connext DDS core libraries and can be used as starting points when configuring QoS for your Connext DDS applications. There are two provided libraries, BuiltinQosLib and BuiltinQosLibExp, which contain different profiles. You can use any of these profiles as base profiles when creating your own XML configurations or simply use these profiles directly in the DDS_*_create_*_with_profile() APIs. There is also a BuiltinQosSnippetLib library, which contains profile "snippets" that can be overlaid upon the profiles to provide additional modifications to your QoS. See 19.3.3.2 QoS Profile Composition on page 858 for more information.

There are three types of built-in profiles:

- **Baseline.X.X.X** profiles represent the QoS defaults for Connext DDS version X.X.X. The defaults for the latest Connext DDS version can be accessed using the BuiltinQosLib::Baseline profile.

- **Generic.X** profiles allow you to easily configure different features and communication use-cases with Connext DDS. For example, there is a Generic.StrictReliable profile for use when your application has a requirement for no data loss, regardless of the application domain.
• **Pattern.X** profiles inherit from **Generic.X** profiles and allow you to configure various domain-specific communication use cases. For example, there is a **Pattern.Alarm** profile that can be used to manage the generation and consumption of alarm events.

The **USER_QOS_PROFILES.xml** file generated by *RTI Code Generator* contains a profile that inherits from the **BuiltInQosLibExp::Generic.StrictReliable** profile as an example of how to use these profiles in your own application.

Example use-cases for these profiles:

- To quickly enable *RTI Monitoring Library* by inheriting from the **BuiltInQosLib::Generic.Monitoring.Common** profile. (See note below.)
- To easily revert to the default QoS values from a previous *Connext DDS* version by inheriting from the correct **BuiltInQosLib::Baseline.X.X.X** profile.
- To set up common use-case configurations and patterns such as strict reliability or large data communication by inheriting from one of the **BuiltInQosLibExp::Generic.X** or **Pattern.X** profiles.

To see the contents of the built-in QoS profiles:

In `<NDDSHOME>/resource/xml`, you will find:

- **BaselineRoot.documentationONLY.xml**—This file contains the root baseline QoS profile corresponding to the default values of *Connext DDS 5.0.0*.
- **BuiltinProfiles.documentationONLY.xml**—This file contains the rest of the built-in QoS profiles.

**Notes:**

- The built-in QoS profiles that enable *RTI Monitoring Library* set the property `rti.monitor.create_function`. Consequently, they only work in *Connext DDS* applications in which the monitoring library can be loaded dynamically. Specifically, the built-in monitoring profiles will not work in these situations:
  
  - When the *Connext DDS* application links the monitoring libraries statically
  - When using a VxWorks platform with Java\(^1\)

For more information, see *Part 9: RTI Monitoring Library* on page 1223.

\(^1\)VxWorks Java platforms require custom supported libraries. To see whether a VxWorks platform with Java is supported, consult the *RTI Connext DDS Core Libraries Platform Notes.*
19.3.2 Overwriting Default QoS Values

There are two ways to overwrite the default QoS used for new entities with values from a profile: programmatically and with an XML attribute.

- You can overwrite the default QoS programmatically with `set_default_<entity>_qos_with_profile` () (where `<entity>` is participant, topic, publisher, subscriber, datawriter, or datareader)
- You can overwrite the default QoS using the XML attribute `is_default_qos` with the `<qos_profile>` tag
- Only for the DomainParticipantFactory: You can overwrite the default QoS using the XML attribute `is_default_participant_factory_profile`. This attribute has precedence over `is_default_qos` if both are set.

In the following example, the `DataWriter` and `DataReader` default QoS will be overwritten with the values specified in a profile named ‘StrictReliableCommunicationProfile’:

```xml
<qos_profile name="StrictReliableCommunicationProfile"
  is_default_qos="true">
  <datawriter_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datawriter_qos>
  <datareader_qos>
    <history>
      <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    <reliability>
      <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
  </datareader_qos>
</qos_profile>
```

If multiple profiles are configured to overwrite the default QoS, only the last one parsed applies.

**Example:**

In this example, the profile used to configure the default QoSs will be `StrictReliableCommunicationProfile`.

```xml
<qos_profile name="BestEffortCommunicationProfile"
  is_default_qos="true">
  ...
</qos_profile>
```
19.3.3 QoS Profile Inheritance and Composition

An individual QoS Profile or Entity QoS (e.g., <datawriter_qos>) can inherit values from other QoS Profiles, and/or be composed out of QoS Snippets. In inheritance, a **base_name** attribute is used to inherit from a single, previously loaded QoS Profile. With composition, a `<base_name>` tag is used to specify a list of one or more QoS Snippets to overlay upon the base profile, creating a new composed profile. The following sections describe how these methods can be used, including best practices. See also 19.6 XML File Syntax on page 885.

### 19.3.3.1 QoS Profile Inheritance

An individual QoS Profile can inherit values from other QoS Profiles described in the XML file by using the attribute **base_name**.

A QoS Profile may also inherit values from other QoS Profiles described in *different* XML files. A QoS Profile can only inherit from other QoS Profiles that have already been loaded. The order in which XML resources are loaded is described in 19.5 How to Load XML-Specified QoS Settings on page 882.

The following examples show how to inherit from other profiles:

**Inheritance Example 1:**

```xml
<qos_library name="Library">
    <qos_profile name="BaseProfile">
        <datawriter_qos>
            ...
        </datawriter_qos>
        <datareader_qos>
            ...
        </datareader_qos>
    </qos_profile>

    <!-- use the base_name attribute to inherit from another profile -->
    <qos_profile name="DerivedProfile" base_name="BaseProfile">
        <datawriter_qos>
            <batch>
                <enable>true</enable>
                <max_samples>100</max_samples>
                <max_data_bytes>LENGTH_UNLIMITED</max_data_bytes>
            </batch>
        </datawriter_qos>
        <datareader_qos>
            ...
        </datareader_qos>
    </qos_profile>
</qos_library>
```
In this example, the QoS Profile called DerivedProfile is constructed via inheritance from the QoS Profile BaseProfile. The profile DerivedProfile inherits BaseProfile by referencing the base profile in the qos_profile attribute base_name="BaseProfile". This means that the datawriter_qos and datareader_qos in DerivedProfile inherit their values from the corresponding datawriter_qos and datareader_qos in BaseProfile. The QoS Profile DerivedProfile first initializes all its QoS policies with the values obtained from BaseProfile. Then it applies any QoS policies explicitly listed in its own definition to override the initialized values. In this example, MyDerivedProfile only modifies the BatchQos policy on the DataWriter QoS.

If a QoS Profile definition does not specify the base_name attribute, then it is initialized from the built-in defaults provided by Connext DDS. See 19.3.1 Built-in QoS Profiles on page 851.

**Inheritance Example 2:**

```xml
<qos_library name="Library">
  <datareader_qos name="BaseProfile">
    ...
  </datareader_qos>
  <datareader_qos name="DerivedProfile" base_name="BaseProfile">
    ...
  </datareader_qos>
</qos_library>
```

The datareader_qos in DerivedProfile inherits its values from the datareader_qos of BaseProfile. In this example, the datareader_qos definition is a shortcut for a QoS Profile definition with a single QoS.

**Inheritance Example 3:**

```xml
<qos_library name="Library">
  <qos_profile name="Profile1">
    <datawriter_qos name="BaseWriterQoS">
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
  <qos_profile name="Profile2">
    <datawriter_qos name="DerivedWriterQos" base_name="Profile1::BaseWriterQos">
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
</qos_library>
```

The datawriter_qos in Profile2 inherits its values from the datawriter_qos in Profile1. The datareader_qos in Profile2 will not inherit the values from the corresponding QoS in Profile1. Since Profile2 doesn’t inherit from any other QoS Profile, the datareader_qos values will be taken from the built-in defaults. See 19.3.1 Built-in QoS Profiles on page 851.
19.3.3 QoS Profile Inheritance and Composition

Inheritance Example 4:

```xml
<qos_library name="Library">
  <qos_profile name="Profile1">
    <datawriter_qos>
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
  <qos_profile name="Profile2">
    <datawriter_qos name="BaseWriterQoS">
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
  <qos_profile name="Profile3" base_name="Profile1">
    <datawriter_qos name="DerivedWriterQos" base_name="Profile2::BaseWriterQos">
      ...
    </datawriter_qos>
    <datareader_qos>
      ...
    </datareader_qos>
  </qos_profile>
</qos_library>
```

The datawriter_qos in Profile3 inherits its values from the datawriter_qos in Profile2. The datareader_qos in Profile3 inherits its values from the datareader_qos in Profile1.

Inheritance Example 5:

```xml
<qos_library name="Library">
  <datareader_qos name="BaseProfile">
    ...
  </datareader_qos>
  <profile name="DerivedProfile" base_name="BaseProfile">
    <datareader_qos>
      ...
    </datareader_qos>
  </profile>
</qos_library>
```

The datareader_qos in DerivedProfile inherits its values from the datareader_qos in BaseProfile.

Inheritance Example 6:

```
Global_QoS.xml
```

```
  <qos_library name="GlobalLibrary">
    <qos_profile name="GlobalProfileA">
    </qos_profile>
  </qos_library>
```

```
Component_QoS.xml
```

```
  <qos_library name="GlobalLibrary">
    <qos_profile name="GlobalProfileA">
    </qos_profile>
  </qos_library>
```
Previous examples show that a QoS Profile or QoS can inherit values from other QoS Profiles or QoSes, which should already be loaded. In this example, a QoS Profile inherits values from another QoS Profile defined in a separate QoS Library, in another file. This is a typical use case where QoSes are constructed by separating them into multiple files. In this example, Global_QoS.xml has to be loaded before Component_QoS.xml.

To learn more about how to load multiple files in your application, see 19.5 How to Load XML-Specified QoS Settings on page 882.

### 19.3.3.1 Limitations of QoS Profile Inheritance

While useful, initializing a QoS Profile from a single base QoS Profile can also be limiting. For example, assume you have the configuration shown in Figure 19.1: Single Inheritance Example below.

**Figure 19.1: Single Inheritance Example**

If you wanted to incorporate monitoring into the QoS Profiles app_1 and app_2, the only option with inheritance would be to create two new QoS Profiles, each inheriting from app_1 and app_2 respectively, and to copy the monitoring XML configuration into each of the two new QoS Profiles as shown in Figure 19.2: Duplication of Configuration in Inheritance on the next page. This results in significant XML code duplication and leads to maintainability issues.
The following section describes how to handle the above scenario using QoS Profile composition.

### 19.3.3.2 QoS Profile Composition

QoS Profile composition uses QoS Snippets to more easily update profiles that you use or inherit. QoS Snippets are small pieces of well-formed, reusable XML QoS that configure a single aspect of QoS, such as enabling monitoring or security.

In the previous example, you could add the monitoring configuration to the new QoS Profiles `app_mon_1` and `app_mon_2` by referring to a QoS Snippet that configures monitoring.
QoS Snippets are intended to be composed into other QoS Snippets and QoS Profiles. As shown in the example below, the syntax used to define a QoS Snippet is the same as that of a QoS Profile, but the intent and usage are different.

The following is an example of the syntax used to define and use QoS Snippets.

**Composition Example 1:**

```xml
<!-- This is a QoS Snippet -->
<qos_profile name="Snippet1">
  <datareader_qos>
    <reliability>
      <kind>RELIABLE_RELIABILITY</kind>
    </reliability>
  </datareader_qos>
</qos_profile>

<!-- This is a QoS Snippet -->
<qos_profile name="Snippet2">
  <datareader_qos>
    <durability>
      <kind>TRANSIENT_LOCAL_DURABILITY</kind>
    </durability>
  </datareader_qos>
</qos_profile>

<qos_profile name="Profile1">
```

19.3.3 QoS Profile Inheritance and Composition

In this example, a QoS Profile inherits from another QoS Profile and uses composition to weave in policies from two QoS Snippets. Specifically, MyDerivedAndComposedProfile is constructed by inheriting from Profile1, then by overlaying Snippet1 and Snippet2. Finally, MyDerivedAndComposedProfile applies its own QoS policies, which overwrite any others. See also 19.3.3.2.2 Order and Precedence of Inheritance on the next page.

It is recommended to use fully qualified names in the element tag if there is ambiguity in the QoS Profile or QoS Snippet names you have loaded in your application.

19.3.3.2.1 How Inheritance and Composition Work Together

The process of inheriting QoS Profiles and composing from QoS Snippets works as follows:

1. The QoS policies are initialized from those in the base profile, using the base_name attribute of the <qos_profile> tag. If the base_name attribute is not present, then the policies are initialized from the builtin defaults defined by Connext DDS.

2. The policies are overridden with those defined in the QoS Snippets listed inside the <base_name> XML tag. The QoS Snippets are applied in the order in which they appear. So the first QoS Snippet (Snippet1 in the example above) overrides the policies that were set from the inherited base QoS Profile (Profile1 in the example), the second QoS Snippet (Snippet2 in the example) overrides whatever was the result of applying Snippet1, and so on.

3. The policies that appear explicitly as elements in the QoS Profile are applied. These override the policies set by the base QoS Profile and the QoS Snippets. In this example, a KEEP_LAST_HISTORY_DEPTH of 6 overrides whatever was set by the base QoS Profile and the QoS Snippets.
You inherit a QoS Profile, but overlay one or more Qos Snippets. Inherit a QoS Profile because you want to subsume the complete definition of the QoS policies for a particular use case. Overlay QoS Snippets onto a QoS Profile so that you override only a single aspect of QoS: for instance, only what is logically associated with monitoring.

### 19.3.3.2.2 Order and Precedence of Inheritance

Values are inherited from the specified elements in the `<base_name>` tag, in order from top to bottom. Values inherited from elements lower in the order (Snippet2 in the examples) will overwrite the same values (if present) from elements higher up (Snippet1 in the examples). Remember that the QoS, QoS Profile, or QoS Snippet should already be loaded as a part of your XML file. (See [19.5 How to Load XML-Specified QoS Settings on page 882](#)).

In the following example, MyDerivedAndComposedProfile inherits from Profile1, keeping Profile1’s SampleDataWriter_A but getting `<durability>` and `<reliability>` from the Snippets rather than from Profile1. Finally, MyDerivedAndComposedProfile applies its own local `<history>` policies.

**Composition Example 2:**

```xml
<qos_profile name="Snippet1">
    <datareader_qos>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
    </datareader_qos>
</qos_profile>

<qos_profile name="Snippet2">
    <datareader_qos>
        <durability>
            <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
        </durability>
    </datareader_qos>
</qos_profile>

<qos_profile name="Profile1">
    <datawriter_qos>
        <publication_name>
            <name>SampleDataWriter_A</name>
        </publication_name>
    </datawriter_qos>
    <datareader_qos>
        <durability>
            <kind>VOLATILE_DURABILITY_QOS</kind>
        </durability>
        <reliability>
            <kind>BEST_EFFORT_RELIABILITY_QOS</kind>
        </reliability>
    </datareader_qos>
</qos_profile>
```
19.3.3 QoS Profile Inheritance and Composition

The final values in MyDerivedAndComposedProfile will be as follows (map the colors in the example to what actually gets used), as shown here:

Composition Example 3

Imagine that Example 2 had the following Snippets instead:
In this example, Snippet2’s `nanosec` overwrites Snippet1's. But since Snippet2 does not specify a `sec`, Snippet1's `sec` is used. The resultant QoS is a combination of the two `reliability` policies:

Imagine now that the QoS Snippets in the above example were reversed, and Snippet2 was listed first in the file. Snippet2 would apply a `nanosec` of 1000000; then Snippet1 would overwrite that with 0 and apply its `sec` of 5. The result would be a `max_blocking_time` of 5 seconds and 0 nanoseconds.

You can use the `rtixmloutpututility` utility to see what the final QoS values will be in your system when composition and inheritance complete their derivations. See 19.3.3.5 Viewing Resolved QoS Values on page 869.

**Composition Example 4:**

If you specify `<base_name>` for a QoS Profile and also specify `<base_name>` for a QoS within it, the `<base_name>` tag or attribute in the QoS will take precedence. That is, `<base_name>` from the QoS Profile will be ignored for the QoS specifying its own `<base_name>`.

The following example illustrates this concept:

```xml
<dds>
  <qos_library>
    <!-- This is a QoS Snippet -->
    <qos_profile name="Snippet2">
      <datawriter_qos>
        <reliability>
          <kind>RELIABLE_RELIABILITY_QOS</kind>
          <max_blocking_time>
            <nanosec>1000000</nanosec>
          </max_blocking_time>
        </reliability>
      </datawriter_qos>
    </qos_profile>
    <!-- This QoS Profile definition uses the Snippets -->
    <qos_profile name="MyDerivedAndComposedProfile" base_name="Profile1">
      <base_name>
        <element>Snippet1</element>
        <element>Snippet2</element>
      </base_name>
    </qos_profile>
  </qos_library>
</dds>
```
19.3.3 QoS Profile Inheritance and Composition

```xml
<qos_profile name="ParentProfile">
    <base_name>
        <element>A</element>
        <element>B</element>
    </base_name>
    ...
    <datawriter_qos name="DW_QoS">
        <base_name>
            <element>C</element>
            <element>D</element>
        </base_name>
    </datawriter_qos>
</qos_profile>
</qos_library>
```

In this example, since DW_QoS has its own list for the `<base_name>` tag, DW_QoS will only inherit values from C and D. It will NOT inherit anything from A and B specified as a part of ParentProfile, since its own `<base_name>` tag overrides it.

### 19.3.3.3 Best Practices for Inheritance and Composition

XML QoS Profile inheritance and composition provide a powerful way to define configurations, allowing flexibility and reusability. It is important to understand the underlying mechanics and follow the best practices described below to maximize usability and avoid unexpected results.

- **Differentiate between QoS Profiles and QoS Snippets.**
  - Think of QoS Profiles as complete definitions of all QoS policies for a particular use case. Construct QoS Profiles so that all aspects of the use case are covered.
  - Think of QoS Snippets as small, generic, orthogonal chunks of QoS policies. Construct QoS Snippets to configure a single aspect of a configuration, such as monitoring or security.

- **Use QoS Profiles for inheritance only, never composition.** Use a QoS Profile in a `<base_name>` attribute, never inside a `<base_name>` element.

- **Use QoS Snippets for composition, never inheritance.** Use a QoS Snippet inside a `<base_name>` element, never in a `<base_name>` attribute.

- **Use QoS Profiles, not QoS Snippets, to create DDS Entities.** Do not pass a QoS Snippet name to the DDS operations `create_<entity>_with_profile()`, `get_<entity>_qos_from_profile()`, `set_qos_with_profile()`, or `set_default_profile()`.

- **Keep QoS Snippets generic and reusable.** Never use the `<topic_filter>` element in a QoS Snippet.

These best practices are illustrated in the following figure and further described in the sections that follow.
In Figure 19.4: Best Practices for Inheritance and Composition above, imagine the results produced by the dotted box, as already illustrated in the previous examples. These results are inherited by QoS Profile 3. QoS Profile 3's snippets are then applied. (QoS Snippet 5 inherits from two other snippets first.) Finally, any policies in QoS Profile 3 that differ from the results produced by the inheritance from profiles 1 and 2 are applied.

Another way to look at Figure 19.4: Best Practices for Inheritance and Composition above is as a tree whose nodes are applied in this order, where "QP" refers to the QoS Profiles in the figure and "QS" refers to the QoS Snippets in the figure:

1. QP1 (because inheritance says we start all the way back at the first inherited profile)
2. QS1, then QS2 (because snippets are overlaid next)
3. QP2 (because it may have deltas that overwrite what has been composed so far)
4. QS3, then QS4 (because QS5 inherits from QS3 and QS4 first)
5. QS5 (because it may have deltas that overwrite QS3 and QS4)
6. QP3 (because it may have deltas that overwrite everything composed so far)
19.3.3.3.1 Differentiate between QoS Profiles and QoS Snippets

When defining a QoS Profile, decide whether you are:

- Creating a QoS Profile intended to create DDS Entities and/or fully define their QoS.
- Creating a QoS Snippet intended as a reusable block to be composed in the definition of QoS Profiles and other QoS Snippets.

These two options are fundamentally different.

A QoS Profile is intended to define the QoS policies used to create a DDS Entity. Therefore, it should match a specific application use case (e.g., sending alarms or streaming periodic data). Moreover, because the QoS Profile will be used to create a DDS Entity, it implicitly defines values for all the QoS Policies that apply to the entity.

When defining a QoS Profile, choose the builtin QoS Profile that most closely matches your use case. Use that builtin QoS Profile as a base profile. For example:

```
<qos_profile name="MyProfile" base_name="BuiltinQosLibExp::Pattern.AlarmEvent">
  <!-- modify the profile by composing with QoS Snippets -->
  <!-- modify the profile by overriding the QoS policies explicitly -->
</qos_profile>
```

Give the QoS Profile a name that makes clear its intended use, as well as the fact that it is a QoS Profile (instead of a QoS Snippet). For example, use “Profile” as a suffix in the name of the QoS Profile or some other consistent naming convention.

A QoS Snippet is intended as a generic block of QoS policies for composition into QoS Profiles and other QoS Snippets. For example, configuring monitoring, configuring Security, and configuring a FlowController are good uses for QoS Snippets.

QoS Snippets should focus on a single aspect of QoS policy and try not to set unrelated policies. This maximizes composability, avoiding interfering with policies set by other QoS Snippets.

QoS Snippets should be generic and reusable across systems and deployments. Therefore, it does not make sense to constrain their applicability using the `<topic_filter>` element within their definition. Doing so may also result in conflict with topic filters set on QoS Profiles that use those QoS Snippets.

Give the QoS Snippet a name that makes clear its intended use, as well as the fact that it is a QoS Snippet (not a regular QoS Profile). For example, use “Snippet” as a suffix in the name of the QoS Snippet or some other consistent naming convention.

19.3.3.3.2 Use QoS Profiles for inheritance only, never composition

Aside from its use for creating DDS Entities, a QoS Profile may be used as the base definition of another QoS Profile. For example:

```
<qos_profile name="MyDerivedProfile" base_name="MyBaseProfile">
  ...
</qos_profile>
```
When used for inheritance, the derived profile is initialized with the policies of the base profile.

A profile should never be used for composition. That is, it should not be referenced within the `<base_name>` element:

```xml
<qos_profile name="MyDerivedProfile">
  <base_name>
    <element>MyBaseProfile</element> <!-- never do this -->
  </base_name>
</qos_profile>
```

Because a QoS Profile implicitly defines all the QoS policies, using it for composition would have the unintended effect of potentially overriding all the policies.

### 19.3.3.3 Use QoS Snippets for composition, never inheritance

QoS Snippets are small pieces of well-formed XML QoS intended to configure a single aspect of a QoS. The proper way to use them is for the composition of other QoS Profiles and QoS Snippets. Therefore, they must only appear within the `<base_name>` tag "element." For example:

```xml
<qos_profile name="MyComposedProfile" base_name="MyBaseProfile">
  <base_name>
    <element>Snippet1</element>
    <element>Snippet2</element>
    <element>Snippet3</element>
  </base_name>
  ...
</qos_profile>

<qos_profile name="MyComposedSnippet" base_name="Snippet1">
  <base_name>
    <element>Snippet2</element>
    <element>Snippet3</element>
  </base_name>
  ...
</qos_profile>
```

Do not use a QoS Snippet for inheritance. For example

```xml
<!-- do not do this -->
<qos_profile name="MyComposedProfile" base_name="Snippet1">
  <base_name>
    <element>Snippet2</element>
    <element>Snippet3</element>
  </base_name>
  ...
</qos_profile>
```
If you use a QoS Snippet for inheritance (i.e., for initializing another Qos Profile), you are using something that was not intended to be a full definition; thus, it may overlook the proper configuration of certain policies for your system.

### 19.3.3.3.4 Use QoS Profiles, not QoS Snippets, to create DDS Entities

The QoS configuration of DDS *Entities* can be specified using QoS Profiles. This is a convenient mechanism that allows separation of configuration from the functional logic of your application.

The *Connext DDS* API contains several operations that reference QoS Profiles by name, such as `create_participant_with_profile()` and `create_topic_with_profile()`. These operations are used to either create DDS *Entities* with the QoS policies referenced by the profile name, or to initialize the *Entity* QoS structure with the QoS policies referenced by the profile. Either way, these operations should not be called using a QoS Snippet name as the reference.

### 19.3.3.5 Keep QoS Snippets generic and reusable

QoS Snippets should be developed with reuse in mind and should not use the `<topic_filter>` element within the definition of the QoS Snippet.

```xml
<!-- do not do this -->
<qos_profile name="MySnippet">
  <datawriter_qos topic_filter="Alarm">
    <reliability>
      <kind>RELIABLE_RELIABILITY</kind>
    </reliability>
  </datawriter_qos>
  <datawriter_qos topic_filter="SensorUpdate">
    <reliability>
      <kind>BEST EFFORTS_RELIABILITY</kind>
    </reliability>
  </datawriter_qos>
  ...
</qos_profile>
```

The `<topic_filter>` element conditionally defines the QoS Profile depending on the *Topic* name associated with the *Entity* being created or configured. Since the QoS Snippet is not intended to create or configure DDS *Entities* directly, it does not make sense to use the `<topic_filter>` element in its definition.

### 19.3.3.4 Enforcement of QoS Profile and QoS Snippet Conventions

*Connext DDS* uses the same syntax for the creation of QoS Profiles and QoS Snippets. Therefore, it does not enforce the conventions described here. Although *Connext DDS* will not detect or prevent violation of these conventions (e.g., if you use a QoS Profile for composition), following these conventions is strongly encouraged to avoid unexpected results. Furthermore, future versions of *Connext DDS* may introduce different syntax that allows differentiating QoS Profiles from QoS Snippets and enforces the conventions. If you follow these conventions now, you can continue using them without violating future syntax.
19.3.3.5 Viewing Resolved QoS Values

The final value for a QoS configuration, especially when using inheritance and QoS Snippet composition, can be visualized at runtime in a variety of ways:

- Locally in your application, the QoS `to_string` functions allow `Entity` QoS objects to be converted into strings and printed, so that you can see the current QoS being used. `Entity` QoS types are `DataReaderQos`, `DataWriterQos`, `PublisherQos`, `SubscriberQos`, `TopicQos`, `DomainParticipantQos` and `DomainParticipantFactoryQos`.

- Additionally, when an entity is created, or when the `set_qos` operation is called on an entity, the QoS settings it is using are output to the log, if logging is configured with a verbosity of `NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL` and category of `NDDS_CONFIG_LOGCATEGORY_API`. (See 25.2 Controlling Messages from Connext DDS on page 984.) If the `DDS_EntityNameQosPolicy` is set, the names will be printed as part of a header to help associate logged QoS settings with the appropriate entities. Connext DDS automatically prints the QoSes of these entities to the log in XML format. Note it is not required that your QoS was configured in XML, it will always be logged in XML format to the log.

The logged QoS when using logging, or the `to_string` functions, will show only the QoS settings that are different from the documented default (several `to_string` overloads can override this behavior). The documented default refers to the default value of a policy as specified by the API reference HTML documentation.

- Remotely, using `RTI Monitor`.

- Remotely, using `RTI Admin Console`. Note that when visualizing the QoS using `Admin Console`, only a subset of the QoS are shown. Only QoS policies that are required for matching are propagated to `Admin Console`.

Here is an example of a `to_string` function in the Modern C++ API:

```cpp
using namespace rti::all;

DataWriterQos the_qos = writer.qos();

// Obtain a string representation of the DataWriterQos object
// Only differences with respect to the documented default will be included
std::string the_string = to_string(the_qos);

// Create another DataWriterQos object and change some policies
DataWriterQos other_qos;
other_qos << Reliability::BestEffort();
// The differences with respect to the other_qos object will now be stored to the string
the_string = to_string(the_qos, other_qos);

// Finally, we can print the entire QoS object (not just differences)
the_string = to_string(the_qos, rti::core::qos_print_all);
```
For older releases, or where code change/recompilation isn't possible, you can use `rtixmloutpututility` to visualize the end result of your QoS settings at entity creation time.

`rtixmloutpututility` allows you to see the final QoS values your entities will receive after inheritance and composition are resolved. Here is an example usage of this utility:

```
$ ./rtixmloutpututility
   -qosFile '/home/xxx/Documents/Tests/CORE-9446/USER_QOS_PROFILES.xml;/home/xxx/Documents/Tests/CORE-1375/USER_QOS_PROFILES.xml'
   -profilePath Data_Library::Data_Profile
   -outputFile Dummy.txt
   -qosTag domain_participant_qos/property
```

To get this utility, including more information about its options and usage, please see: [https://github.com/rticomunity/rticonnextdds-xml-output-utility](https://github.com/rticomunity/rticonnextdds-xml-output-utility).

19.3.4 Topic Filters

A QoS profile may contain several writer, reader and topic QoSs. Connext DDS will select a QoS based on the evaluation of a filter expression (as defined in the POSIX fnmatch API (1003.2-1992 Section B.6)) on the topic name. The filter expression is specified as an attribute in the XML QoS definition. For example:

```
<qos_profile name="StrictReliableCommunicationProfile"
   <datawriter_qos topic_filter="A*"/>
   <datawriter_qos topic_filter="B*"/>
```

If `topic_filter` is not specified in a QoS, Connext DDS will assume the filter `*`. The QoSs with an explicit `topic_filter` attribute definition will be evaluated in order; they have precedence over a QoS without a
topic_filter expression.

The topic_filter attribute is only used with the following APIs:

DomainParticipantFactory:

- `get_<entity>_qos_from_profile_w_topic_name()` (where `<entity>` may be topic, datareader, or datareader; see 9.2.5 Getting QoS Values from a QoS Profile on page 608)

DomainParticipant:

- `create_datawriter_with_profile()` (see 7.3.1 Creating DataWriters on page 292)
- `create_datareader_with_profile()` (see 8.3.1 Creating DataReaders on page 510)
- `create_topic_with_profile()` (see 5.1.1 Creating Topics on page 217)

Publisher:

- `create_datawriter_with_profile()` (see 7.3.1 Creating DataWriters on page 292)

Subscriber:

- `create_datareader_with_profile()` (see 8.3.1 Creating DataReaders on page 510)

Topic:

- `set_qos_with_profile()` (see 5.1.3 Setting Topic QoS Policies on page 219)

DataWriter:

- `set_qos_with_profile()` (see 7.2.4.3 Changing QoS Settings After the Publisher Has Been Created on page 281)

DataReader:

- `set_qos_with_profile()` (see 8.3.9 Setting DataReader QoS Policies on page 543)

Note: in the Modern C++ API, use `dds::core::QosProvider::<entity>_qos_w_topic_name()` to obtain the Qos associated with a topic. For example:

```cpp
auto reader_qos = qos_provider.extensions().datareader_qos_w_topic_name("Example Topic");
dds::sub::DataReader<Foo> reader(subscriber, topic, reader_qos);
```
Other APIs will ignore QoSs with a `topic_filter` value different than "*". A QoS Profile with QoSs using `topic_filter` can also inherit from other QoS Profiles. In this case, inheritance will consider the value of the `topic_filter` expression.

**Example 1:**

```xml
<qos_library name="Library">
  <qos_profile name="BaseProfile">
    <datawriter_qos>
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T1*">
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T2*">
      ...
    </datawriter_qos>
  </qos_profile>
  <qos_profile name="DerivedProfile" base_name="BaseProfile">
    <datawriter_qos topic_filter="T11">
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T21">
      ...
    </datawriter_qos>
    <datawriter_qos topic_filter="T31">
      ...
    </datawriter_qos>
  </qos_profile>
</qos_library>
```

The `datawriter_qos` with `topic_filter` T11 in DerivedProfile will inherit its values from the `datawriter_qos` with `topic_filter` T1* in BaseProfile. The `datawriter_qos` with `topic_filter` T21 in DerivedProfile will inherit its values from the `datawriter_qos` with `topic_filter` T2* in BaseProfile. The `datawriter_qos` with `topic_filter` T31 in DerivedProfile will inherit its values from the `datawriter_qos` without `topic_filter` in BaseProfile.
Example 2:

```xml
<qos_library name="Library">
  <qos_profile name="BaseProfile">
    <datawriter_qos topic_filter="T1*"/>
    ...
  </datawriter_qos>
  <datawriter_qos name="T2DataWriterQoS" topic_filter="T2*"/>
  ...
</datawriter_qos>
</qos_profile>
<qos_profile name="DerivedProfile" base_name="BaseProfile">
  <datawriter_qos topic_filter="T11" base_name="BaseProfile::T2DataWriterQoS"/>
  ...
  <datawriter_qos topic_filter="T21"/>
  ...
</datawriter_qos>
</qos_profile>
</qos_library>
```

Although the `topic_filter` expressions do not match, the `datawriter_qos` with `topic_filter` T11 in DerivedProfile will inherit its values from the `datawriter_qos` with `topic_filter` T2* in BaseProfile. `topic_filter` is not used with inheritance from QoS to QoS. The `datawriter_qos` with `topic_filter` T21 in DerivedProfile will inherit its values from the `datawriter_qos` with `topic_filter` T2* in BaseProfile.

Example 3:

```xml
<qos_library name="Library">
  <datawriter_qos name="BaseQos" topic_filter="T1"/>
  ...
</datawriter_qos>
  <datawriter_qos name="DerivedQos" base_name="BaseQos" topic_filter="T2"/>
  ...
</datawriter_qos>
</qos_library>
```

In the case of a single QoS profile, although the `topic_filter` expressions do not match, the `datawriter_qos` named DerivedQos with `topic_filter` T2 will inherit its values from the `datawriter_qos` named BaseQos with `topic_filter` T1.

**Important Note About Topic Filters**

Use the `topic_filter` attribute with caution. In most cases, governance of QoS is improved by using discrete, named QoS profiles with no more than one of each kind of entity QoS section (`datareader_qos`, `datawriter_qos`, etc.) in each profile. If the `topic_filter` attribute contains a typographical error or omission, it is possible for a topic not to match the intended filter expression. This can result in, for example, the entity being silently assigned the default QoS. The `topic_filter` attribute may be preferred in cases where wildcards are used extensively to reduce duplication in the XML. In these cases, the resulting QoS of each entity should be independently and empirically confirmed. Tools that can help confirm an entity’s QoS are
RTI Monitor and (as described in 19.3.3.5 Viewing Resolved QoS Values on page 869) the rtixm-loutpututility.

19.3.5 QoS Profiles with a Single QoS

The definition of an individual QoS outside a profile is a shortcut for defining a QoS profile with a single QoS. For example:

```xml
<datawriter_qos name="KeepAllWriter">
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</datawriter_qos>
```

is equivalent to:

```xml
<qos_profile name="KeepAllWriter">
    <datawriter_qos>
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
    </datawriter_qos>
</qos_profile>
```

19.4 Tags for Configuring QoS with XML

To configure the QoS for an Entity using XML, use the following tags:

- `<participant_factory_qos>`
  
  **Note:** The only QoS policies that can be configured for the DomainParticipantFactory are `<entity_factory>` and `<logging>`.

- `<domain_participant_qos>`
- `<publisher_qos>`
- `<subscriber_qos>`
- `<topic_qos>`
- `<datawriter_qos>` or `<writer_qos>` (writer_qos is valid only with DTD validation)
- `<datareader_qos>` or `<reader_qos>` (reader_qos is valid only with DTD validation)

Each QoS can be identified by a name. The QoS can inherit its values from other QoSs described in the XML file. For example:

```xml
<datawriter_qos name="DerivedWriterQos" base_name="Lib::BaseWriterQos">
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</datawriter_qos>
```
In the above example, the datawriter_qos named 'DerivedWriterQos' inherits the values from 'BaseWriterQos' in the library 'Lib'. The HistoryQosPolicy **kind** is set to KEEP_ALL_HISTORY_QOS.

Each XML tag with an associated name can be uniquely identified by its fully qualified name in C++ style.

The writer, reader and topic QoSs can also contain an attribute called **topic_filter** that will be used to associate a set of topics to a specific QoS when that QoS is part of a QoS profile. See 19.3.4 Topic Filters on page 870 and 19.3 QoS Profiles on page 850.

### 19.4.1 QosPolicies

The fields in a QosPolicy are described in XML using a 1-to-1 mapping with the equivalent C representation. For example, the Reliability QosPolicy is represented with the following C structures:

```c
struct DDS_Duration_t {
    DDS_Long sec;
    DDSUnsignedLong nanosec;
}

struct DDS_ReliabilityQosPolicy {
    DDS_ReliabilityQosPolicyKind kind;
    DDS_Duration_t max_blocking_time;
}
```

The equivalent representation in XML is as follows:

```xml
<reliability>
    <kind></kind>
    <max_blocking_time>
        <sec></sec>
        <nanosec></nanosec>
    </max_blocking_time>
</reliability>
```

### 19.4.2 Sequences

In general, sequences in QosPolicies are described with the following XML format:

```xml
<a_sequence_member_name>
    <element>...</element>
    <element>...</element>
    ...
</a_sequence_member_name>
```

Each element of the sequence is enclosed in an <element> tag. For example:

```xml
<property>
    <value>
        <element>
            <name>my name</name>
            <value>my value</value>
        </element>
    </value>
</property>
```
A sequence without elements represents a sequence of length 0. For example:

```xml
<discovery>
  <!-- initial_peers sequence contains zero elements -->
  <initial_peers/>
</discovery>
```

For sequences that may have a default initialization that is not empty (such as the initial_peers field in the DISCOVERY QosPolicy (DDS Extension) on page 640), using the above construct would result in an empty list and not the default value. So to simply show a sequence for the sake of completeness, but not change its default value, comment it out, as follows:

```xml
<discovery>
  <!-- initial_peers sequence contains the default value -->
  <!-- <initial_peers/> -->
</discovery>
```

As a general rule, sequences defined in a derived QoS will replace the corresponding sequences in the base QoS. (The concepts of derived and base QoS are described in 19.3.3 QoS Profile Inheritance and Composition on page 854.) For example, consider the following:

```xml
<qos_profile name="MyBaseProfile">
  <domain_participant_qos>
    <discovery>
      <initial_peers>
        <element>192.168.1.1</element>
        <element>192.168.1.2</element>
      </initial_peers>
    </discovery>
  </domain_participant_qos>
</qos_profile>

<qos_profile name="MyDerivedProfile" base_name="MyBaseProfile">
  <domain_participant_qos>
    <discovery>
      <initial_peers>
        <element>192.168.1.3</element>
      </initial_peers>
    </discovery>
  </domain_participant_qos>
</qos_profile>
```

The initial peers sequence defined above in the participant QoS of MyDerivedProfile will contain a single element with a value 192.168.1.3. The elements 192.168.1.1 and 192.168.1.2 will not be inherited. However, there is one exception to this behavior.

The <property> and <data_tags> tags provide an attribute called **inherit** that allows you to choose the inheritance behavior for the sequence defined within the tag.
By default, the value of the attribute `inherit` is true. Therefore, the `<property>` tag defined within a derived QoS profile will inherit its elements from the `<property>` tag defined within a base QoS profile.

In the following example, the property sequence defined in the participant QoS of MyDerivedProfile will contain two properties:

- `dds.transport.UDPv4.builtin.send_socket_buffer_size` will be inherited from the base profile and have the value 524288.
- `dds.transport.UDPv4.builtin.recv_socket_buffer_size` will overwrite the value defined in the base QoS profile with 1048576.

```xml
<qos_profile name="MyBaseProfile">
  <domain_participant_qos>
    <property>
      <value>
        <element>
          <name>.dds.transport.UDPv4.builtin.send_socket_buffer_size</name>
          <value>524288</value>
        </element>
        <element>
          <name>dds.transport.UDPv4.builtin.recv_socket_buffer_size</name>
          <value>2097152</value>
        </element>
      </value>
    </property>
  </domain_participant_qos>
</qos_profile>

<qos_profile name="MyDerivedProfile" base_name="MyBaseProfile">
  <domain_participant_qos>
    <property>
      <value>
        <element>
          <name>dds.transport.UDPv4.builtin.recv_socket_buffer_size</name>
          <value>1048576</value>
        </element>
      </value>
    </property>
  </domain_participant_qos>
</qos_profile>
```

To discard all the properties defined in the base QoS profile, set `inherit` to false.

In the following example, the property sequence defined in the participant QoS of MyDerivedProfile will contain a single property named `dds.transport.UDPv4.builtin.recv_socket_buffer_size`, with a value of 1048576. The property `dds.transport.UDPv4.builtin.send_socket_buffer_size` will not be inherited.
19.4.3 Arrays

In general, the arrays contained in the QosPolicies are described with the following XML format:

```xml
<an_array_member_name>
  <element>...</element>
  <element>...</element>
  ...
</an_array_member_name>
```

Each element of the array is enclosed in an `<element>` tag.

As a special case, arrays of octets are represented with a single XML tag enclosing an array of decimal/hexadecimal values between 0..255 separated with commas.

For example:

```xml
<reader_qos>
  ...
  <protocol>
    <virtual_guid>
```
19.4.4 Enumeration Values

Enumeration values are usually represented using the programming language name without the DDS_ or NDDS_ prefix. For example:

```xml
<value>
  1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16
</value>
</virtual_guid>
</protocol>
</reader_qos>
```

You can get the actual enumeration values by using the XSD file distributed with Connext DDS. See 19.9.2 XML File Validation During Editing on page 890. (Note: the XSD file provides stricter validation and better auto-completion than the corresponding DTD file.)

19.4.5 Time Values (Durations)

You can use the following special values for fields that require seconds or nanoseconds:

- DURATION_INFINITE_SEC
- DURATION_ZERO_SEC
- DURATION_INFINITE_NSEC
- DURATION_ZERO_NSEC
- DURATION_AUTO_SEC
- DURATION_AUTO_NSEC

For example:

```xml
<deadline>
  <period>
    <sec>DURATION_INFINITE_SEC</sec>
    <nanosec>DURATION_INFINITE_NSEC</nanosec>
  </period>
</deadline>
```

These values are the same as the programming language names without the DDS_ or NDDS_ prefix. You can find these special values by using the XSD file distributed with Connext DDS. See 19.9.2 XML File Validation During Editing on page 890. (Note: the XSD file provides stricter validation and better auto-completion than the corresponding DTD file.)
19.4.6 Transport Properties

You can configure transport plugins using the DomainParticipant's 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.

- Properties for the builtin transports are described in 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy on page 801. You can also set these properties in XML using the <transport_builtin> tag.
- Properties for other transport plugins such as RTI TCP Transport¹ are described in their respective chapters in this manual.

This example configures the builtin UDPv4 transport using the <transport_builtin> tags:

```xml
<domain_participant_qos>
  <transport_builtin>
    <udp4>
      <message_size_max>1024</message_size_max>
    </udp4>
  </transport_builtin>
</domain_participant_qos>
```

You can do the same thing with the UDPv6 (<udpv6>) and SHMEM (<shmem>) transports (and, if RTI Real-Time WAN Transport is installed, with UDPv4_WAN (<udp4_wan>)). For example:

```xml
<domain_participant_qos>
  <transport_builtin>
    <udpv6>
      <message_size_max>1024</message_size_max>
    </udpv6>
  </transport_builtin>
</domain_participant_qos>
```

You cannot use the <transport_builtin> tag for the other transport plugins, like TCP, DTLS, WAN, LBRTPS, and ZRTPS. (You must use the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 properties for those.)

The following example configures the builtin UDPv4 transport using the legacy approach, via regular XML tags; you can find the names of these 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 properties in the Property Reference Guide:

```xml
<domain_participant_qos>
  <property>
    <value>
      <element>
        <name>dds.transport.UDPv4.builtin.parent.message_size_max</name>
        <value>65507</value>
      </element>
    </value>
  </property>
</domain_participant_qos>
```

¹RTI TCP Transport is included with Connext DDS, but is not enabled by default.
Some of the properties in the Property Reference Guide are described as "promoted" to <transport_builtin> tag usage. For example, dds.transport.UDPv4.builtin.send_socket_buffer_size can be entered either in a <name> element as shown above or via the <transport_builtin> tag as shown here:

```xml
<domain_participant_qos>
  <transport_builtin>
    <udpv4>
      <send_socket_buffer_size>131072</send_socket_buffer_size>
    </udpv4>
  </transport_builtin>
</domain_participant_qos>
```

All of the UDPv4, UDPv4_WAN (if installed), UDPv6, and SHMEM transport properties (except those that are private or deprecated) have been "promoted" for use in a <transport_builtin> tag, if desired; however, the legacy way of using the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 properties (as described in the Property Reference Guide) is still supported. The <transport_builtin> tag is simply an easier way to configure your transports when using XML, for those properties that support it.

Note that programmatically, you must use the properties. For example:

```c
retCode = DDS_PropertyQosPolicyHelper_add_property(
    &participantQos.property,
    "dds.transport.UDPv4.builtin.parent.message_size_max",
    "5000",
    DDS_BOOLEAN_FALSE);
```

If you happen to set a transport property both ways—via the PROPERTY QoS <name> and via the <transport_builtin> tag—the <transport_builtin> method takes precedence. For example:

```xml
<qos_profile name="Precedence_Test_Tag_Properties">
  <participant_qos>
    <transport_builtin>
      <mask>SHMEM</mask>
      <shmem>
        <address_bit_count>0</address_bit_count>
      </shmem>
    </transport_builtin>
    <property>
      <value/>
      <!-- SHMEM -->
      <element>
        <name>dds.transport.shmem.builtin.parent.address_bit_count</name>
```

```xml
</element>
</property>
</participant_qos>
```

```xml
</qos_profile>
```
In this example, the `address_bit_count` will be 0.

Deprecated or private properties have no XML QoS representation via the `<transport_builtin>` tag. For example, `dds.transport.UDPv4.builtin.ignore_nonup_interfaces`, although it is a UDPv4 transport plugin property, cannot be specified via the `<transport_builtin>` tag, because it has been deprecated.

### 19.4.7 Thread Settings

See Table 21.1 XML Tags for ThreadSettings_t.

### 19.4.8 Entity Names

The `name` and `role_name` fields in the 7.5.11 ENTITY_NAME QosPolicy (DDS Extension) on page 418 have three distinct possible values: NULL, an empty string, and a non-empty string. Each of these three states are specified in XML in a different way.

To specify that the `name` or `role_name` of an entity is NULL, use the `xsi:nil` attribute. The `xsi:nil` attribute can be set to either "true" or "false". For example, to set the participant name to NULL:

```
<participant_name>
  <name xsi:nil="true">
  </name>
</participant_name>
```

To specify the empty string, leave the XML element empty:

```
<participant_name>
  <name/>
</participant_name>
```

To specify a non-empty string:

```
<participant_name>
  <name>"My Participant's Name"</name>
</participant_name>
```

### 19.5 How to Load XML-Specified QoS Settings

There are several ways to load XML QoS profiles into your application. In C, Traditional C++, Java and .NET, it's the singleton DomainParticipantFactory that loads these profiles. Applications using the Modern C++ API can create any number of instances of `dds::core::QosProvider` with different parameters to load different QoS profiles or, they can use the singleton `QosProvider::Default()`.

Here are the various approaches, listed in load order:
19.5 How to Load XML-Specified QoS Settings

- `<NDDSHOME>/resource/xml/NDDS_QOS_PROFILES.xml`
  This file is loaded automatically if it exists (not the default) and ignore_resource_profile in the 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633 is FALSE (the default). (First to be loaded)

  Note: NDDS_QOS_PROFILES.xml does not exist by default. However, NDDS_QOS_PROFILES.example.xml is shipped with the host bundle of the product; you can copy it to NDDS_QOS_PROFILES.xml and modify it for your own use. The file contains the default QoS values that will be used for all entity kinds.

- **XML files in NDDS_QOS_PROFILES**
  One or more XML files separated by semicolons referenced by the environment variable NDDS_QOS_PROFILES are loaded automatically if they exist and ignore_environment_profile in 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633 is FALSE (the default).

  Semicolons indicate to Connext DDS to load multiple files or strings all at once. For example:

  On Linux and macOS systems, with bash:

  ```bash
  export NDDS_QOS_PROFILES='file:///usr/local/default_dds_1.xml; file:///usr/local/default_dds_2.xml'
  ```

  From a Windows command prompt:

  ```cmd
  set NDDS_QOS_PROFILES=file://D:/Data/ConnextDDSExample/default_dds_1.xml; file://D:/Data/ConnextDDSExample/default_dds_2.xml
  ```

- `<working directory>/NDDS_QOS_PROFILES.xml`
  This file is loaded automatically if it exists and ignore_user_profile in 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633 is FALSE (the default).

- `<working directory>/USER_QOS_PROFILES.xml`
  This file is loaded automatically if it exists and ignore_user_profile in 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633 is FALSE (the default).

- **XML files in url_profile**
  One or more XML files referenced by url_profile (in 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633) will be loaded automatically if specified.

- **XML strings in string_profile**
  The sequence of XML strings referenced by string_profile (in 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633) will be loaded automatically if specified. See 19.7 XML String Syntax on page 888. (Last to be loaded)

  Note: The url_profile and string_profile fields are useful for adding profiles programmatically, when you do not want to use an environment variable.

You may use a combination of the above approaches.
The location of the XML documents (only files and strings are supported) is specified using URL (Uniform Resource Locator) format. See 19.8 URL Groups (Loading Redundant Locations) on page 889. For example:

- File Specification: file:///usr/local/default.dds.xml
- String Specification: str://"<dds><qos_library>…</qos_library></dds>"

If you omit the URL schema name, Connext DDS will assume a file name. For example:

- File Specification: /usr/local/default.dds.xml

**Note:** The path you specify can be absolute or relative. If you specify a relative path, it should be a path to a file that is lower down in the file hierarchy, not higher up.

Duplicate QoS profiles are not allowed. Connext DDS will report an error message in these scenarios. To overwrite a QoS profile, use 19.3.3 QoS Profile Inheritance and Composition on page 854.

Several QoS profiles are built into the Connext DDS core libraries and can be used as starting points when configuring QoS for your Connext DDS applications. For details, see 19.4 Tags for Configuring QoS with XML on page 874.

To load redundant locations for a single XML file, see 19.8 URL Groups (Loading Redundant Locations) on page 889.

### 19.5.1 Loading, Reloading and Unloading Profiles

You do not have to explicitly call `load_profiles()`. QoS profiles are loaded when any of these DomainParticipantFactory operations are called:

- `create_participant()` (see 9.3.1 Creating a DomainParticipant on page 615)
- `create_participant_with_profile()` (see 9.3.1 Creating a DomainParticipant on page 615)
- `get_<entity>_qos_from_profile()` (where `<entity>` is `participant`, `topic`, `publisher`, `subscriber`, `datawriter`, or `datareader`) (see 9.2.5 Getting QoS Values from a QoS Profile on page 608)
- `get_<entity>_qos_from_profile_w_topic_name()` (where `<entity>` is `topic`, `datawriter`, or `datareader`) (see 9.2.5 Getting QoS Values from a QoS Profile on page 608)
- `get_default_participant_qos()` (see 9.2.2 Getting and Setting Default QoS for DomainParticipants on page 607)
- `get_qos_profile_libraries()` (See 19.10.1 Retrieving a List of Available Libraries on page 894)
- `get_qos_profiles()` (See 19.4 Tags for Configuring QoS with XML on page 874)
- `load_profiles()`
In the Modern C++ API, the previous operations cause the default QosProvider (QosProvider::Default()) to load the QoS profiles. Any other QosProvider that an application instantiates will load the QoS Profiles it is configured to load in its constructor.

QoS profiles are reloaded when either of these DomainParticipantFactory operations are called:

- `set_default_participant_qos_with_profile()` (see 9.2.2 Getting and Setting Default QoS for DomainParticipants on page 607)
- `set_default_library()` (see 7.2.4.4 Getting and Setting the Publisher’s Default QoS Profile and Library on page 282)
- `set_default_profile()` (see 7.2.4.4 Getting and Setting the Publisher’s Default QoS Profile and Library on page 282)

It is important to distinguish between loading and reloading:

- **Loading** only happens when there are no previously loaded profiles. This could be when the profiles are loaded the first time or after a call to `unload_profiles()`.
- **Reloading** replaces all previously loaded profiles. Reloading a profile does not change the QoS of entities that have already been created with previously loaded profiles.

The DomainParticipantFactory also has an `unload_profiles()` operation that frees the resources associated with the XML QoS profiles.

19.6 XML File Syntax

The contents of the XML configuration file must follow an important hierarchy: the file contains one or more libraries; each library contains one or more profiles; each profile contains QoS settings.

In addition, the file must follow these syntax rules:

- The syntax is XML and the character encoding is UTF-8.
- Opening tags are enclosed in `<>`, closing tags are enclosed in `</>`.
19.6.1 Using Environment Variables in XML

- A tag value is a UTF-8 encoded string. Legal values are alphanumeric characters. The middleware’s parser will remove all leading and trailing spaces\(^a\) from the string before it is processed.
- For example, `<tag> value </tag>` is the same as `<tag>value</tag>`.
- All values are case-sensitive unless otherwise stated.
- Comments are enclosed as follows: `<!-- comment -->`.
- The root tag of the configuration file must be `<dds>` and end with `</dds>`.
- The primitive types for tag values are specified in Table 19.1 Supported Tag Values.

### Table 19.1 Supported Tag Values

<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>true, false</td>
<td>Not case-sensitive</td>
</tr>
<tr>
<td>DDS_Enum</td>
<td>A string. Legal values are those listed in the API Reference HTML document for the C or Java API.</td>
<td>Must be specified as a string. (Do not use numeric values.)</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>-2147483648 to 2147483647 or 0x80000000 to 0x7fffffff or LENGTH_UNLIMITED or DDS_LENGTH_UNLIMITED</td>
<td>A 32-bit signed integer</td>
</tr>
<tr>
<td>DDSUnsignedLong</td>
<td>0 to 4294967296 or 0 to 0xffffffff</td>
<td>A 32-bit unsigned integer</td>
</tr>
<tr>
<td>String</td>
<td>UTF-8 character string</td>
<td>All leading and trailing spaces are ignored between two tags</td>
</tr>
</tbody>
</table>

See also:

- 19.6.1 Using Environment Variables in XML below
- 19.6.2 Using Special Characters in XML on the next page
- 19.6.3 Specifying Fully Qualified Names in XML on page 888

### 19.6.1 Using Environment Variables in XML

The text within an XML tag and attribute can refer to environment variable. To do so, use the following notation:

\(^a\)Leading and trailing spaces in enumeration fields will not be considered valid if you use the distributed XSD document to do validation at run-time with a code editor.

886
For example:

```xml
<element attr="The attribute is $(MY_ATTRIBUTE)">
    <name>The name is $(MY_NAME)</name>
    <value>The value is $(MY_VALUE)</value>
</element>
```

When the Connext DDS XML parser parses the above tags, it will replace the references to environment variables with their actual values.

### 19.6.2 Using Special Characters in XML

In the XML QoS configuration file, you may sometimes want to use special characters for the name of an element. If so, escape them by surrounding the attribute value with `&quot;` symbols in XML. The `&quot;` symbol is a special escape character within the XML standard that represents double quotes (").

For example:

```xml
<qos_profile name="&quot;&lt;MySpecial::NameProfile&gt;&quot;">
    <base_name>
        <element>Data_Library::&quot;&lt;MySpecial::NameProfile&gt;&quot;</element>
    </base_name>
</qos_profile>
```

The name of this profile is `<MySpecial::NameProfile>`. It contains special characters like `<`, `>` and `:` within its name. When not escaped, `"::"` is often used as a name separator when referring to an element in the XML hierarchy. (See 19.6.3 Specifying Fully Qualified Names in XML on the next page.)

You can use the same scheme when inheriting from the QoS profile in the `base_name` attribute or the `<base_name>` tag.

For example:

```xml
<qos_profile name="SpecialNameDerived"
    is_default_qos="true">
    <base_name>
        <element>Data_Library::&quot;&lt;MySpecial::NameProfile&gt;&quot;</element>
    </base_name>
</qos_profile>
```

In this example, the profile is contained within the `Data_Library` QoS Library.

This idea is applicable to all tags that can perform inheritance (any tag that has a `base_name` attribute or can contain the `<base_name>` tag).
19.6.3 Specifying Fully Qualified Names in XML

When specifying a parent to inherit from in the `base_name` attribute or the `<base_name>` tag of a QoS policy in an XML file, you can refer to elements using a fully qualified naming scheme. This causes the search to begin from the root of the XML Document Object Model (DOM) tree parsed by the XML parser.

For that you need to follow this scheme:

```
<parent_tag_name>::<child_tag_name>::<grandchild_tag_name> . . .
```

The resulting name should match the declaration hierarchy. The declaration hierarchy represents the ordering of the tags as described in the XML schema. For QoS configuration, the hierarchy is as follows:

```
<qos_library_name>::<qos_profile_name>
```

Here "::" is the path separator in the XML DOM tree. Specifying a fully qualified name is useful when you want to refer to elements within another `<qos_library>` tag.

For example:

```
<qos_library name="Data_Library">
  <qos_profile name="Data_Profile" base_name="BuiltinQosLib::Generic.StrictReliable">
    . . .
  </qos_profile>
</qos_library>
```

19.7 XML String Syntax

XML profiles can be described using strings. This configuration is useful for architectures without a file system.

There are two different ways to configure `Entities` via XML strings:

- String URLs are prefixed by the URI schema `str://` and enclosed in double quotes. For example:

  ```
  str://"<dds><qos_library>...</qos_library></dds>"
  ```

  The string URLs can be specified in the environment variable `NDDS_QOS_PROFILES` as well as in the field `url_profile` in 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633. Each string URL must contain a whole XML document.

- The `string_profile` field in the 9.4.2 PROFILE QosPolicy (DDS Extension) on page 633 allows you to split an XML document into multiple strings. For example:
const char * MyXML[4] =
{
    "<dds>",
    "<qos_library name="MyLibrary">",
    "</qos_library>",
    "</dds>
};
factoryQos.profile.string_profile.from_array(MyXML,4);

Only one XML document can be specified with the string_profile field.

19.8 URL Groups (Loading Redundant Locations)

Use URL groups to specify multiple locations for a single XML file, to provide redundancy and fault tolerance. Specify the locations by enclosing them in square brackets. The syntax of a URL group is:

[URL1 | URL2 | URL2 | ... | URLn]

You can specify either an XML file or an XML string. For example:

[file:///usr/local/default_dds.xml | file:///usr/local/alternative_default_dds.xml | str://"<dds><qos_library name="Data_Library"><qos_profile name="Data_Profile" base_name="BuiltinQosLibExp::Generic.StrictReliable" is_default_qos="true" /></qos_library></dds>"

The OR operand (|) tells Connext DDS to load one file/string at a time, starting from the left. If the first file/string is not available, Connext DDS tries loading the next one, and so on. For example, if the first location loaded successfully, Connext DDS does not load the subsequent locations. Brackets are not required for URL groups with a single file/string. In fact, single XML files separated by semicolons are URL groups without brackets.

Here’s an example (on a Linux system with bash) that loads multiple files, including one bracketed URL group with redundant file locations:

export NDDS_QOS_PROFILES="[file:///usr/local/default_dds.xml | file:///usr/local/alternative_default_dds.xml]; file:///usr/local/default_dds_2.xml"

See also 19.5 How to Load XML-Specified QoS Settings on page 882 for information on loading multiple XML files.

19.9 How the XML is Validated

19.9.1 Validation at Run-Time

Connext DDS validates the input XML files using a builtin Document Type Definition (DTD).

You can find a copy of the builtin DTD in <NDDSHOME>/resource/schema/rti_dds_qos_profiles.dtd. (This is only a copy of what the Connext DDS core uses. Changing this file has no effect unless you specify its path with the <!DOCTYPE> tag, described below.)

You can overwrite the builtin DTD by using the XML tag, <!DOCTYPE>. For example, the following indicates that Connext DDS must use a DTD file from a user’s directory to perform validation:
The DTD path can be absolute, or relative to the application's current working directory.

If the specified file does not exist, you will see the following error:

```
RTIXMLDtdParser_parse: !open DTD file
```

If you do not specify the DOCTYPE tag in the XML file, the built-in DTD is used.

The XML files used by Connext DDS can be versioned using the attribute version in the <dds> tag. For example:

```
<dds version="6.1.1">
  ...
</dds>
```

Although the attribute version is not required during the validation process, it helps to detect DTD incompatibility scenarios by providing better error messages.

For example, if an application using Connext DDS 6.1.1 tries to load an XML file from Connext DDS 4.5z and there is some incompatibility in the XML content, the following parsing error will be printed:

```
ATTENTION: The version declared in this file (4.5z) is different from the version of Connext DDS (6.1.1).
If these versions are not compatible, that incompatibility could be the cause of this error.
```

### 19.9.2 XML File Validation During Editing

*Connext DDS* provides DTD and XSD files that describe the format of the XML content. We recommend including a reference to one of these documents in the XML file that contains the QoS profiles—this provides helpful features in code editors such as Visual Studio and Eclipse, including validation and auto-completion while you are editing the XML file.

The DTD and XSD definitions of the XML elements are in

```
<NDDSHOME>/resource/schema/rti_dds_qos_profiles.dtd
```

and

```
<NDDSHOME>/resource/schema/rti_dds_qos_profiles.xsd
```

respectively. (*<NDDSHOME>* is described in Paths Mentioned in Documentation on page 1.)

To include a reference to the XSD document in your XML file, use the attribute

```
xsi:noNamespaceSchemaLocation
```

in the <dds> tag. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="<NDDSHOME>/resource/schema/rti_dds_qos_profiles.xsd">
  ...
</dds>
```

To include a reference to the DTD document in your XML file use the <!DOCTYPE> tag. For example:
We recommend including a reference to the XSD file in the XML documents because it provides stricter validation and better auto-completion than the corresponding DTD file.

19.10 Using QoS Profiles in Your *Connext DDS* Application

You can use the operations listed in Table 19.2 *Operations for Working with QoS Profiles* to refer to and use QoS profiles (see 19.3 *QoS Profiles on page 850*) described in XML files and XML strings.

**Table 19.2 Operations for Working with QoS Profiles**

<table>
<thead>
<tr>
<th>Working With ...</th>
<th>Profile-Related Operations</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataReaders</td>
<td>set_qos_with_profile</td>
<td>8.3.9.3 Changing QoS Settings After the DataReader has been Created on page 548</td>
</tr>
<tr>
<td>DataWriters</td>
<td>set_qos_with_profile</td>
<td>7.3.15.3 Changing QoS Settings After the DataWriter Has Been Created on page 338</td>
</tr>
</tbody>
</table>
### Table 19.2 Operations for Working with QoS Profiles

<table>
<thead>
<tr>
<th>Working With ...</th>
<th>Profile-Related Operations</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>DomainParticipants</td>
<td>create_datareader_with_profile</td>
<td>8.3.1 Creating DataReaders on page 510</td>
</tr>
<tr>
<td></td>
<td>create_datawriter_with_profile</td>
<td>7.3.1 Creating DataWriters on page 292</td>
</tr>
<tr>
<td></td>
<td>create_publisher_with_profile</td>
<td>7.2.2 Creating Publishers on page 276</td>
</tr>
<tr>
<td></td>
<td>create_subscriber_with_profile</td>
<td>8.2.2 Creating Subscribers on page 491</td>
</tr>
<tr>
<td></td>
<td>create_topic_with_profile</td>
<td>5.1.1 Creating Topics on page 217</td>
</tr>
<tr>
<td></td>
<td>get_default_library</td>
<td>9.3.7.4 Getting and Setting DomainParticipant's Default QoS Profile and Library on page 626</td>
</tr>
<tr>
<td></td>
<td>get_default_profile</td>
<td>9.3.7.4 Getting and Setting DomainParticipant's Default QoS Profile and Library on page 626</td>
</tr>
<tr>
<td></td>
<td>get_default_profile_library</td>
<td>9.3.7.4 Getting and Setting DomainParticipant's Default QoS Profile and Library on page 626</td>
</tr>
<tr>
<td></td>
<td>set_default_datareader_qos_with_profile</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_datawriter_qos_with_profile</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_library</td>
<td>9.3.7.4 Getting and Setting DomainParticipant's Default QoS Profile and Library on page 626</td>
</tr>
<tr>
<td></td>
<td>set_default_profile</td>
<td>9.3.7.4 Getting and Setting DomainParticipant's Default QoS Profile and Library on page 626</td>
</tr>
<tr>
<td></td>
<td>set_default_publisher_qos_with_profile</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_subscriber_qos_with_profile</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_default_topic_qos_with_profile</td>
<td>9.3.7.5 Getting and Setting Default QoS for Child Entities on page 627</td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>9.3.7.3 Changing QoS Settings After DomainParticipant Has Been Created on page 625</td>
</tr>
</tbody>
</table>
### Table 19.2 Operations for Working with QoS Profiles

<table>
<thead>
<tr>
<th>Working With ...</th>
<th>Profile-Related Operations</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DomainParticipantFactory</strong></td>
<td>create_participant_with_profile</td>
<td>9.3.1 Creating a DomainParticipant on page 615</td>
</tr>
<tr>
<td></td>
<td>get_datareader_qos_from_profile</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_datawriter_qos_from_profile</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_datawriter_qos_from_profile_w_topic_name</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_datareader_qos_from_profile_w_topic_name</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_default_library</td>
<td>9.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library on page 606</td>
</tr>
<tr>
<td></td>
<td>get_default_profile</td>
<td>9.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library on page 606</td>
</tr>
<tr>
<td></td>
<td>get_default_profile_library</td>
<td>9.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library on page 606</td>
</tr>
<tr>
<td></td>
<td>get_participant_qos_from_profile</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_publisher_qos_from_profile</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_subscriber_qos_from_profile</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_topic_qos_from_profile</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_topic_qos_from_profile_w_topic_name</td>
<td>9.2.5 Getting QoS Values from a QoS Profile on page 608</td>
</tr>
<tr>
<td></td>
<td>get_qos_profiles</td>
<td>19.10.2 Retrieving a List of Available QoS Profiles on page 895</td>
</tr>
<tr>
<td></td>
<td>get_qos_profile_libraries</td>
<td>19.10.1 Retrieving a List of Available Libraries on the next page</td>
</tr>
<tr>
<td></td>
<td>load_profiles</td>
<td>19.5.1 Loading, Reloading and Unloading Profiles on page 884</td>
</tr>
<tr>
<td></td>
<td>reload_profiles</td>
<td>19.5.1 Loading, Reloading and Unloading Profiles on page 884</td>
</tr>
<tr>
<td></td>
<td>set_default_participant_qos_with_profile</td>
<td>9.2.2 Getting and Setting Default QoS for DomainParticipants on page 607</td>
</tr>
<tr>
<td></td>
<td>set_default_library</td>
<td>9.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library on page 606</td>
</tr>
<tr>
<td></td>
<td>set_default_profile</td>
<td>9.2.1.1 Getting and Setting the DomainParticipantFactory's Default QoS Profile and Library on page 606</td>
</tr>
<tr>
<td></td>
<td>unload_profiles</td>
<td>19.5.1 Loading, Reloading and Unloading Profiles on page 884</td>
</tr>
</tbody>
</table>
Table 19.2 Operations for Working with QoS Profiles

<table>
<thead>
<tr>
<th>Working With</th>
<th>Profile-Related Operations</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publishers</td>
<td>create_datawriter_with_profile</td>
<td>7.2.2 Creating Publishers on page 276</td>
</tr>
<tr>
<td></td>
<td>get_default_library</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_profile</td>
<td>7.2.4 Getting and Setting the Publisher's Default QoS Profile and Library on page 282</td>
</tr>
<tr>
<td></td>
<td>get_default_profile_library</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_datawriter_qos_with_profile</td>
<td>7.2.4.5 Getting and Setting Default QoS for DataWriters on page 282</td>
</tr>
<tr>
<td></td>
<td>set_default_library</td>
<td>7.2.4 Getting and Setting the Publisher's Default QoS Profile and Library on page 282</td>
</tr>
<tr>
<td></td>
<td>set_default_profile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>7.2.4.3 Changing QoS Settings After the Publisher Has Been Created on page 281</td>
</tr>
<tr>
<td>Subscribers</td>
<td>create_datareader_with_profile</td>
<td>8.3.1 Creating DataReaders on page 510</td>
</tr>
<tr>
<td></td>
<td>get_default_library</td>
<td></td>
</tr>
<tr>
<td></td>
<td>get_default_profile</td>
<td>8.2.4 Getting Settings Subscriber's Default QoS Profile and Library on page 497</td>
</tr>
<tr>
<td></td>
<td>get_default_profile_library</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_default_datareader_qos_with_profile</td>
<td>8.2.4.5 Getting and Setting Default QoS for DataReaders on page 498</td>
</tr>
<tr>
<td></td>
<td>set_default_library</td>
<td>8.2.4.4 Getting Settings Subscriber's Default QoS Profile and Library on page 497</td>
</tr>
<tr>
<td></td>
<td>set_default_profile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_qos_with_profile</td>
<td>8.2.4.3 Changing QoS Settings After Subscriber Has Been Created on page 496</td>
</tr>
<tr>
<td>Topics</td>
<td>set_qos_with_profile</td>
<td>5.1.3 Setting Topic QosPolicies on page 219</td>
</tr>
</tbody>
</table>

Note: For the Modern C++ API, please refer to the RTI Connext DDS API Reference HTML documentation, Configuring QoS Profiles with XML.

19.10.1 Retrieving a List of Available Libraries

To get a list of available QoS libraries, call the DomainParticipantFactory’s get_qos_profile_libraries() operation, which returns the names of all QoS libraries that have been loaded by Connext DDS.
19.10.2 Retrieving a List of Available QoS Profiles

To get a list of available QoS profiles, call the DomainParticipantFactory’s `get_qos_profiles()` operation, which returns the names of all profiles within a specified QoS library. Either the input QoS library name must be specified or the default profile library must have been set prior to calling this function.

```
DDS_ReturnCode_t get_qos_profiles (struct DDS_StringSeq *profile_names,
                                  const char *library_name)
```

19.11 Configuring Logging Via XML

Logging can be configured via XML using the DomainParticipantFactory’s LoggingQosPolicy. See 25.2.2 Configuring Logging via XML on page 992 for additional details.
Chapter 20 Multi-channel DataWriters

In Connext DDS, producers publish data to a Topic, identified by a topic name; consumers subscribe to a Topic and optionally to specific content by means of a content-filter expression.

A Market Data Example:

A producer can publish data on the Topic "MarketData" which can be defined as a structured record containing fields that identify the exchange (e.g., "NYSE" or "NASDAQ"), the stock symbol (e.g., "APPL" or "JPM"), volume, bid and ask prices, etc.

Similarly, a consumer may want to subscribe to data on the "MarketData" Topic, but only if the exchange is "NYSE" or the symbol starts with the letter "M." Or the consumer may want all the data from the "NYSE" whose volume exceeds a certain threshold, or may want MarketData for a specific stock symbol, regardless of the exchange, and so on.

The middleware’s efficient implementation of content-filtering is critical for scenarios such as the above "Market Data" example, where there are large numbers of consumers, large volumes of data, or Topics that transmit information about many data-objects or subjects (e.g., individual stocks).

Traditionally, middleware products use four approaches to implement content filtering: Producer-based, Consumer-based, Server-based, and Network Switch-based.

- **Producer-based approaches** push the burden of filtering to the producer side. The producer knows what each consumer wants and delivers to the consumer only the data that matches the consumer's filter. This approach is suitable when using point-to-point protocols such as TCP—it saves bandwidth and lowers the load on the consumer—but it does not work if data is distributed via multicast. Also, this approach does not scale to large numbers of consumers, because the producer would be overburdened by the need to filter for each individual consumer.
20.1 What is a Multi-channel DataWriter?

A Multi-channel DataWriter is a DataWriter that is configured to send data over multiple multicast addresses, according to some filtering criteria applied to the data.

To determine which multicast addresses will be used to send the data, the middleware evaluates a set of filters that are configured for the DataWriter. Each filter "guards" a channel—a set of multicast addresses. Each time a multi-channel DataWriter writes data, the filters are applied. If a filter evaluates to true, the data is sent over that filter’s associated channel (set of multicast addresses). We refer to this type of filter as a Channel Guard filter.
Figure 20.1: Multi-channel Data Flow

20.1 What is a Multi-channel DataWriter?

Figure 20.1: Multi-channel Data Flow

DataWriter

Filter1 → Channel1
Filter2 → Channel2
Filter3 → Channel3

Channel = \{ ChannelMulticastAddress1, port1, ChannelMulticastAddress2, port2, ChannelMulticastAddress3, port3, \ldots \}
20.1 What is a Multi-channel DataWriter?

Figure 20.2: Multi-Channel Evaluation

Multi-channel DataWriters can be used to trade off network bandwidth with the unnecessary processing of unwanted data for situations where there are multiple DataReaders who are interested in different subsets of data that come from the same data stream (Topic). For example, in Financial applications, the data stream may be quotes for different stocks at an exchange. Applications usually only want to receive data (quotes) for only a subset of the stocks being traded. In tracking applications, a data stream may carry information on hundreds or thousands of objects being tracked, but again, applications may only be interested in a subset.

The problem is that the most efficient way to deliver data to multiple applications is to use multicast so that a data value is only sent once on the network for any number of subscribers to the data. However, using multicast, an application will receive all of the data sent and not just the data in which it is interested, thus extra CPU time is wasted to throw away unwanted data. With this QoS, you can analyze the data-usage patterns of your applications and optimize network vs. CPU usage by partitioning the data into multiple multicast streams. While network bandwidth is still being conserved by sending data only once using multicast, most applications will only need to listen to a subset of the multicast addresses and receive a reduced amount of unwanted data.

Note: Your system can gain more of the benefits of using multiple multicast groups if your network uses Layer 2 Ethernet switches. Layer 2 switches can be configured to only route multicast packets to those ports that have added membership to specific multicast groups. Using those switches will ensure that only the multicast packets used by applications on a node are routed to the node; all others are filtered-out by the switch.
20.2 How to Configure a Multi-channel DataWriter

To configure a multi-channel DataWriter, simply define a list of all its channels in the DataWriter's 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432.

Each channel consists of filter criterion to apply to the data and a set of multicast destinations (transport, address, port) that will be used for sending data that matches the filter. You can think of this sequence of channels as a table like the one shown below:

<table>
<thead>
<tr>
<th>If the Data Matches this Filter...</th>
<th>Send the Data to these Multicast Destinations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbol MATCH '[A-K]*'</td>
<td>UDPv4:225.0.0.1:9000</td>
</tr>
<tr>
<td>Symbol MATCH '[L-Q]*'</td>
<td>UDPv4:225.0.0.2:9001</td>
</tr>
<tr>
<td>Symbol MATCH '[P-Z]*'</td>
<td>UDPv4:225.0.0.3:9002;225.0.0.4:9003;</td>
</tr>
</tbody>
</table>

The example C++ code in Figure 20.3: Using the MULTI_CHANNEL QosPolicy below shows how to configure the channels.

Figure 20.3: Using the MULTI_CHANNEL QosPolicy

```cpp
// initialize writer_qos with default values
publisher->get_default_datawriter_qos(writer_qos);
// Initialize MULTI_CHANNEL Qos Policy
// Assign the filter name
// Possible options: DDS_STRINGMATCHFILTER_NAME, DDS_SQLFILTER_NAME
writer_qos.multi_channel.filter_name = (char*) DDS_STRINGMATCHFILTER_NAME;
// Create two channels
writer_qos.multi_channel.channels.ensure_length(2,2);
// First channel
writer_qos.multi_channel.channels[0].filter_expression = DDS_String_dup("Symbol MATCH '^[A-M]''");
writer_qos.multi_channel.channels[0].multicast_settings.ensure_length(1,1);
writer_qos.multi_channel.channels[0].multicast_settings[0].receive_port = 8700;
writer_qos.multi_channel.channels[0].multicast_settings[0].receive_address = DDS_String_dup("239.255.1.1");
// Second channel
writer_qos.multi_channel.channels[1].multicast_settings.ensure_length(1,1);
writer_qos.multi_channel.channels[1].multicast_settings[0].receive_port = 8800;
writer_qos.multi_channel.channels[1].multicast_settings[0].receive_address = DDS_String_dup("239.255.1.2");
writer_qos.multi_channel.channels[1].filter_expression = DDS_String_dup("Symbol MATCH '^[N-Z]''");
```

// Create writer
20.2.1 Limitations

When considering use of a multi-channel DataWriter, please be aware of the following limitations:

- A \textit{DataWriter} that uses the MULTI\_CHANNEL QosPolicy will ignore multicast and unicast addresses specified on the reader side through the 8.6.5 TRANSPORT\_MULTICAST QosPolicy (DDS Extension) on page 591 and 7.5.28 TRANSPORT\_UNICAST QosPolicy (DDS Extension) on page 460. The \textit{DataWriter} will not publish DDS samples on these locators.

- Multi-channel \textit{DataWriters} cannot be configured to use the Durable Writer History feature (described in 13.3 Durable Writer History on page 739).

- Multi-channel \textit{DataWriters} rely on the \texttt{rtps\_object\_id} in the 7.5.5 DATA\_WRITER\_PROTOCOL QosPolicy (DDS Extension) on page 389 to be DDS\_RTPS\_AUTO\_ID (which causes automatic assignment of object IDs to channels).

- To guarantee reliable delivery, a \textit{DataReader}'s 7.4.6 PRESENTATION QosPolicy on page 362 must be set to per-instance ordering (DDS\_INSTANCE\_PRESENTATION\_QOS, the default value), instead of per-topic ordering (DDS\_TOPIC\_PRESENTATION\_QOS), and the matching \textit{DataWriter}'s 7.5.16 MULTI\_CHANNEL QosPolicy (DDS Extension) on page 432 must use expressions that only refer to key fields.

20.3 Multi-Channel Configuration on the Reader Side

\textit{No special changes are required in a subscribing application to get data from a multi-channel DataWriter.}

If you want the \textit{DataReader} to subscribe to only a subset of the channels, use a ContentFilteredTopic, as described in 5.4 ContentFilteredTopics on page 227. For example:

```c
// Create a content filtered topic
contentFilter =
    participant->create_contentfilteredtopic_with_filter(
        "FilteredTopic",
        topic,
        "symbol MATCH 'NYSE/BAC,NASDAQ/MSFT,NASDAQ/GOOG",
        parameters,
        DDS_STRINGMATCHFILTER\_NAME);
// Create a DataReader that uses the content filtered topic
reader = subscriber->create_datareader(contentFilter,
    DDS\_DATAREADER\_QOS\_DEFAULT,
    NULL, 0);
```
From there, Connext DDS takes care of all the necessary steps:

- The DataReader automatically discovers all the DataWriters—including multi-channel DataWriters—for the Topic it subscribes to.
- When the DataReader discovers a multi-channel DataWriter, it also discovers the list of channels used by that DataWriter.
- When the multi-channel DataWriter discovers a DataReader, it also discovers the content filters specified by that DataReader, if any.

With all this information, Connext DDS automatically determines which channels are of "interest" to the DataReader.

A DataReader is interested in a channel if and only if the set of data values for which the channel guard filter evaluates to TRUE intersects the set of data values for which the DataReader's content filter evaluates to TRUE. If a DataReader does not use a content filter, then it is interested in all the channels.

Figure 20.4: Filter Intersection

In this scenario, the DataReader is interested in Channel1 and Channel2, but not Channel3.
Market Data Example, continued:

If the channel guard filter for Channel 1 is 'Symbol MATCH [A-K]*' then the channel will only transfer data for stocks whose symbol starts with a letter in the A to K range.

That is, it will transfer data on 'APPL', 'GOOG', and 'IBM', but not on 'MSFT', 'ORCL', or 'YHOO'. Channel 1 will be of interest to DataReaders whose content filter includes at least one stock whose symbol starts with a letter in the A to K range.

A DataReader that specifies a content filter such as "Symbol MATCH 'IBM, YHOO' " will be interested in Channel1.

A DataReader that specifies a content filter such as "Symbol MATCH '[G-M]*'" will also be interested in Channel1.

A DataReader that specifies a content filter such as "Symbol MATCH '[M-T]*' " will not be interested in Channel1.

20.4 Where Does the Filtering Occur?

If multi-channel DataWriters are used, the filtering can occur in three places:

- 20.4.1 Filtering at the DataWriter below
- 20.4.2 Filtering at the DataReader below
- 20.4.3 Filtering on the Network Hardware on the next page

20.4.1 Filtering at the DataWriter

Each time data is written, the DataWriter evaluates each of the channel guard filters to determine which channels will transmit the data. This filtering occurs on the DataWriter.

Filtering on the DataWriter side is scalable because the number of filter evaluations depends only on the number of channels, not on the number of DataReaders. Usually, the number of channels is smaller than the number of possible DataReaders.

As explained in 20.7 Performance Considerations on page 906, if the channel guard filters are configured to only look at the "key" fields in the data, the channel filtering becomes a very efficient lookup operation.

20.4.2 Filtering at the DataReader

The DataReader will listen on the multicast addresses that correspond to the channels of interest (see Figure 20.3: Using the MULTI_CHANNEL QosPolicy on page 900). When a channel is 'of interest', it means that it is possible for the channel to transmit data that meets the content filter of the DataReader, however the channel may also transmit data that does not pass the DataReader's content filter. Therefore, the DataReader has to filter all incoming data on that channel to determine if it passes its content filter.
Market Data Example, continued:

Channel 1, identified by guard filter "Symbol MATCH '[A-M]*'" , will be of interest to DataReaders whose content filter includes at least one stock whose symbol starts with a letter in the A to K range.

A DataReader with content filter "Symbol MATCH 'GOOG'" will listen on Channel1.

In addition to 'GOOG', the DataReader will also receive DDS samples corresponding to stock symbols such as 'MSFT' and 'APPL'. The DataReader must filter these DDS samples out.

As explained in 20.7 Performance Considerations on page 906, if the DataReader’s content filters are configured to only look at the "key" fields in the data, the DataReader filtering becomes a very efficient lookup operation.

20.4.3 Filtering on the Network Hardware

DataReaders will only listen to multicast addresses that correspond to the channels of interest. The multicast traffic generated in other channels will be filtered out by the network hardware (routers, switches).

Layer 3 routers will only forward multicast traffic to the actual destination ports. However, by default, layer 2 switches treat multicast traffic as broadcast traffic. To take advantage of network filtering with layer 2 devices, they must be configured with IGMP snooping enabled (see 20.7.1 Network-Switch Filtering on page 906).

20.5 Fault Tolerance and Redundancy

To achieve fault tolerance and redundancy, configure the DataWriter’s 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432 to publish a DDS sample over multiple channels or over different multicast addresses within a single channel. Figure 20.5: Using the MULTI_CHANNEL QosPolicy with Overlapping Channels below shows how to use overlapping channels.

If a DDS sample is published to multiple multicast addresses, a DataReader may receive multiple copies of the DDS sample. By default, duplicates are discarded by the DataReader and not provided to the application. To change this default behavior, use the Durable Reader State property, dds.data_reader.state.filter_redundant_samples (see 13.4.4 How To Configure a DataReader for Durable Reader State on page 747).

Figure 20.5: Using the MULTI_CHANNEL QosPolicy with Overlapping Channels

```cpp
// initialize writer_qos with default values
publisher->get_default_datawriter_qos(writer_qos);
// Initialize MULTI_CHANNEL Qos Policy
// Assign the filter name
// Possible options: DDS_STRINGMATCHFILTER_NAME and DDS_SQLFILTER_NAME
writer_qos.multi_channel.filter_name = (char*) DDS_STRINGMATCHFILTER_NAME;
// Create two channels
writer_qos.multi_channel.channels.ensure_length(2,2);
// First channel
```
20.6 Reliability with Multi-Channel DataWriters

20.6.1 Reliable Delivery

Reliable delivery is only guaranteed when the access scope in the Subscriber's 7.4.6 PRESENTATION QosPolicy on page 362 is set to DDS_INSTANCE_PRESENTATION_QOS (default value) and the filters in the DataWriter's 7.5.16 MULTI_CHANNEL QosPolicy (DDS Extension) on page 432 are keyed-only based.

Market Data Example, continued:

Given the following IDL description for our MarketData topic type:

```c
Struct MarketData {
    @key string<255> Symbol;
    double Price;
}
```

A guard filter "Symbol MATCH 'APPL'" is keyed-only based.

A guard filter "Symbol MATCH 'APPL' and Price < 100" is not keyed-only based.

If any of the guard filters are based on non-key fields, Connext DDS only guarantees reception of the most recent data from the multi-channel DataWriter.

20.6.2 Reliable Protocol Considerations

Reliability is maintained on a per-channel basis. Each channel has its own reliability channel send window:

- low_watermark and high_watermark: The low and high watermarks control the send-window levels (when not using batching, this is a number of DDS samples; when using batching, this is a
number of batches) that determine when to switch between regular and fast heartbeat rates (see 7.5.5.1 High and Low Watermarks on page 394). With multi-channel DataWriters, high_watermark and low_watermark are computed from the channel with the smaller send-window size and they apply to all the channels. Therefore, because the watermark is determined by the channel with the smallest send-window, periodic heartbeating cannot be controlled on a per-channel basis.

- **heartbeats_per_max_samples**: This field defines the number of piggyback heartbeats per current send-window. For multi-channel DataWriters, piggyback heartbeats are sent per channel. The send-window size that is used to calculate the piggyback heartbeat rate is the smallest across all channels.

### 20.7 Performance Considerations

#### 20.7.1 Network-Switch Filtering

By default, multicast traffic is treated as broadcast traffic by layer 2 switches. To avoid flooding the network with broadcast traffic and take full advantage of network filtering, the layer 2 switches should be configured to use IGMP snooping. Refer to your switch’s manual for specific instructions.

When IGMP snooping is enabled, a switch can route a multicast packet to just those ports that subscribe to it, as seen in Figure 20.6: IGMP Snooping below.

**Figure 20.6: IGMP Snooping**

![Diagram of IGMP Snooping]

#### 20.7.2 DataWriter and DataReader Filtering

20.4 Where Does the Filtering Occur? on page 903 describes the three places where filtering can occur with Multi-channel DataWriters. To improve performance when filtering occurs on the reader and/or
writer sides, use filter expressions that are only based on keys (see 2.4 DDS Samples, Instances, and Keys on page 18). Then the results of the filter are cached in a hash table on a per-key basis.

Market Data Example, continued:

The filter expressions in the Market Data example are based on the value of the field, Symbol. To make filter operations on this field more efficient, declare Symbol as a key. For example:

```c
struct {
    @key string<MAX_SYMBOL_SIZE> Symbol;
}
```

You can also improve performance by increasing the number of buckets associated with the hash table. To do so, use the instance_hash_buckets field in the 7.5.22 RESOURCE LIMITS QosPolicy on page 451 on both the writer and reader sides. A higher number of buckets will provide better performance, but requires more resources.
Chapter 21 Connext DDS Threading Model

This chapter describes the internal threads that Connext DDS uses for sending and receiving data, maintaining internal state, and calling user code when events occur such as the arrival of new DDS data samples. It may be important for you to understand how these threads may interact with your application.

A DomainParticipant uses three main types of threads:

- **Database Thread** maintains the database of DDS Entities stored in the DomainParticipant. It is responsible for purging the objects marked for deletion when they are no longer needed. See 21.1 Database Thread on the next page.

- **Event Thread** detects triggered events and acts accordingly, invoking user functions when needed (e.g., when a callback was specified for that specific event). See 21.2 Event Thread on page 910.

- **Receive Threads** get bytes from transport plugins, then deserialize and store the (meta)data in the receive queue of a DataReader and invoke the on_data_available() callback. Receive threads are also responsible for processing metadata (e.g., discovery traffic, ACKs, NACKs). See 21.3 Receive Threads on page 911.

The actual number of threads depends on the configuration of various QoS policies as well as the implementation of the transports used by the DomainParticipant to send and receive data. In addition, other threads might be created for specific purposes:

- **Interface Tracking Thread** retrieves new interface data and compares it with the previous value. The thread detects interface changes and notifies the user/application of the changes. It is used in the IP Mobility feature to detect interface changes. You can disable this thread. See 17.1.1.3 Disabling IP Locator Change Propagation on page 828.

- **Transport-Specific Threads** handle the tasks that are specific to a transport (e.g., the TCP Transport plugin creates two threads, one for control and one for events).
• **Asynchronous Publishing Threads** handle the data transmission for Asynchronous Publishers (see 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345).

• **Asynchronous Batch Flushing Threads** handle batches of data samples, flushing them when needed. These threads are only created when batching is enabled (see 7.5.2 BATCH QosPolicy (DDS Extension) on page 374) and \texttt{max\_flush\_delay} is not DURATION\_INFINITE.

• **Topic Query Publication Threads** publish historical samples in response to a TopicQuery. These threads are only created when topic query dispatch is enabled (see 7.5.24 TOPIC_QUERY_DISPATCH QosPolicy (DDS Extension) on page 455).

• **User Threads**: in addition, your application may present threads that are not part of Connext DDS. If those threads call a DDS API, Connext DDS will automatically register them (i.e., allocate some resources to keep statistics and to handle concurrent access to DDS Entities). To free up all the allocated resources, you may need to unregister these threads, as explained in 21.8 Unregistering Threads on page 916.

Through various QosPolicies, the user application can configure the priorities and other properties of the threads created by Connext DDS. In real-time systems, the user often needs to set the priorities of all threads in an application relative to each other for the proper operation of the system.

For information on checking thread names, see 21.9 Identifying Threads Used by Connext DDS on page 916.

### 21.1 Database Thread

Connext DDS uses internal data structures to store information about locally-created and remotely-discovered Entities. In addition, it will store various objects and data used by Connext DDS for maintaining proper communications between applications. This “database” is created for each DomainParticipant.

As Entities and objects are created and deleted during the normal operation of the user application, different entries in the database may be created and deleted as well. Because multiple threads may access objects stored in the database simultaneously, the deletion and removal of an object from the database happens in two phases to support thread safety.

When an entry/object in the database is deleted either through the actions of user code or as a result of a change in system state, it is only marked for deletion. It cannot be actually deleted and removed from the database until Connext DDS can be sure that no threads are still accessing the object. Instead, the actual removal of the object is delegated to an internal thread that Connext DDS spawns to periodically wake up and purge the database of deleted objects.

This thread is known as the Database thread (also referred to as the database cleanup thread).

• Only one Database thread is created for each DomainParticipant.
21.2 Event Thread

The 9.5.1 DATABASE QosPolicy (DDS Extension) on page 637 of the DomainParticipant configures both the resources used by the database as well as the properties of the cleanup thread. Specifically, the user may want to use this QosPolicy to set the priority, stack size and thread options of the cleanup thread. You must set these options before the DomainParticipant is created, because once the cleanup thread is started as a part of participant creation, these properties cannot be changed.

The period at which the database-cleanup thread wakes up to purge deleted objects is also set in the DATABASE QosPolicy. Typically, this period is set to a long time (on the order of a minute) since there is no need to waste CPU cycles to wake up a thread only to find nothing to do.

However, when a DomainParticipant is destroyed, all of the objects created by the DomainParticipant will be destroyed as well. Many of these objects are stored in the database, and thus must be destroyed by the cleanup thread. The DomainParticipant cannot be destroyed until the database is empty and is destroyed itself. Thus, there is a different parameter in the DATABASE QosPolicy, shutdown_cleanup_period, that is used by the database cleanup thread when the DomainParticipant is being destroyed. Typically set to be on the order of a second, this parameter reduces the additional time needed to destroy a DomainParticipant simply due to waiting for the cleanup thread to wake up and purge the database.

21.2 Event Thread

During operation, Connext DDS must wake up at different intervals to check the condition of many different time-triggered or periodic events. These events are usually to determine if something happened or did not happen within a specified time. Often the condition must be checked periodically as long as the Entity for which the condition applies still exists. Also, the DomainParticipant may need to do something periodically to maintain connections with remote Entities.

For example, the 7.5.7 DEADLINE QosPolicy on page 405 is used to ensure that DataWriters have published data or DataReaders have received data within a specified time period. Similarly, the 7.5.15 LIVELINESS QosPolicy on page 427 configures Connext DDS both to check periodically to see if a DataWriter has sent a liveliness message and to send liveliness messages periodically on the behalf of a DataWriter. As a last example, for reliable connections, heartbeats must be sent periodically from the DataWriter to the DataReader so that the DataReader can acknowledge the data that it has received, see Reliable Communications (Chapter 11 on page 688).

Connext DDS uses an internal thread, known as the Event thread, to do the following:

- Check whether or not deadlines have been missed
- Invoke user-installed Listener callbacks to notify the application of missed deadlines
- Send heartbeats to maintain reliable connections

**Note:** Only one Event thread is created per DomainParticipant.

The 9.5.5 EVENT QosPolicy (DDS Extension) on page 662 of the DomainParticipant configures both the properties and resources of the Event thread. Specifically, the user may want to use this QosPolicy to
set the priority, stack size and thread options of the Event thread. You must set these options before the DomainParticipant is created, because once the Event thread is started as a part of participant creation, these properties cannot be changed.

The EVENT QosPolicy also configures the maximum number of events that can be handled by the Event thread. While the Event thread can only service a single event at a time, it must maintain a queue to hold events that are pending. The initial_count and max_count parameters of the QosPolicy set the initial and maximum size of the queue.

The priority of the Event thread should be carefully set with respect to the priorities of the other threads in a system. While many events can tolerate some amount of latency between the time that the event expires and the time that the Event thread services the event, there may be application-specific events that must be handled as soon as possible.

For example, if an application uses the liveliness of a remote DataWriter to infer the correct operation of a remote application, it may be critical for the user code in the DataReader Listener callback, on_liveliness_changed(), to be called by the Event thread as soon as it can be determined that the remote application has died. The operating system uses the priority of the Event thread to schedule this action.

### 21.3 Receive Threads

Connext DDS uses internal threads, known as Receive threads, to process the data packets received via underlying network transports. These data packets may contain meta-traffic exchanged by DomainParticipants for discovery, or user data (and meta-data to support reliable connections) destined for local DataReaders.

As a result of processing packets received by a transport, a Receive thread may respond by sending packets on the network. Discovery packets may be sent to other DomainParticipants in response to ones received. ACK/NACK packets are sent in response to heartbeats to support a reliable connection.

When a DDS sample arrives, the Receive thread is responsible for deserializing and storing the data in the receive queue of a DataReader as well as invoking the on_data_available() DataReaderListener callback (see 8.3.4 Setting Up DataReaderListeners on page 512).

The number of Receive threads that Connext DDS will create for a DomainParticipant depends on how you have configured the QosPolicies of DomainParticipants, DataWriters and DataReaders as well as on the implementation of a particular transport. The behavior of the builtin transports is well specified. However, if a custom transport is installed for a DomainParticipant, you will have to understand how the custom transport works to predict how many Receive threads will be created.

The following discussion applies on a per-transport basis. A single Receive thread will only service a single transport.
Connext DDS will try to create receive resources\(^1\) for every port of every transport on which it is configured to receive messages. The 7.5.28 TRANSPORT–UNICAST QosPolicy (DDS Extension) on page 460 for DomainParticipant, DataWriters, and DataReaders, the 8.6.5 TRANSPORT–MULTICAST QosPolicy (DDS Extension) on page 591 for DataReaders and the 9.5.2 DISCOVERY QosPolicy (DDS Extension) on page 640 for DomainParticipants all configure the number of ports and the number of transports that Connext DDS will try to use for receiving messages.

Generally, transports will require Connext DDS to create a new receive resource for every unique port number. However, this is both dependent on how the underlying physical transport works and the implementation of the transport plug-in used by Connext DDS. Sometimes Connext DDS only needs to create a single receive resource for any number of ports.

When Connext DDS finds that it is configured to receive data on a port for a transport for which it has not already created a receive resource, it will ask the transport if any of the existing receive resources created for the transport can be shared. If so, then Connext DDS will not have to create a new receive resource. If not, then Connext DDS will.

The TRANSPORT–UNICAST, TRANSPORT–MULTICAST, and DISCOVERY QosPolicies allow you customize ports for receiving user data (on a per-DataReader basis) and meta-traffic (DataWriters and DomainParticipants); ports can be also set differently for unicast and multicast.

How do receive resources relate to Receive threads? Connext DDS will create a Receive thread to service every receive resource that is created. If you use a socket analogy, then for every socket created, Connext DDS will use a separate thread to process the data received on that socket.

So how many threads will Connext DDS create by default—using only the built-in UDPv4 and shared memory transports and without modifying any QosPolicies?

Three Receive threads are created for meta-traffic\(^2\):

- 2 for unicast (one for UDPv4, one for shared memory)
- 1 for multicast (for UDPv4)\(^3\)

Two Receive threads created for user data:

- 2 for unicast (UDPv4, shared memory)
- 0 for multicast (because user data is not sent via multicast by default)

---

\(^{1}\)If UDPv4 was the only transport that Connext DDS supports, we would call these receive resources ‘sockets.’

\(^{2}\)Meta-traffic refers to traffic internal to Connext DDS related to dynamic discovery (see Discovery (Chapter 15 on page 764).

\(^{3}\)Multicast is not supported by shared memory transports.
Therefore, by default, you will have a total of five Receive threads per DomainParticipant. By using only a single transport and disabling multicast, a DomainParticipant can have as few as 2 Receive threads.

Similar to the Database and Event threads, a Receive thread is configured by the 9.5.6 RECEIVER_POOL QosPolicy (DDS Extension) on page 664. However, note that the thread properties in the RECEIVER_POOL QosPolicy apply to all Receive threads created for the DomainParticipant.

### 21.4 Exclusive Areas, RTI Connext DDS Threads, and User Listeners

Connext DDS Event and Receive threads may invoke user code through the Listener callbacks installed on different Entities while executing internal Connext DDS code. In turn, user code inside the callbacks may invoke Connext DDS APIs that reenter the internal code space of Connext DDS. For thread safety, Connext DDS allocates and uses mutual exclusion semaphores (mutexes).

As discussed in 4.5 Exclusive Areas (EAs) on page 197, when multiple threads and multiple mutexes are mixed together, deadlock may result. To prevent deadlock from occurring, Connext DDS is designed using careful analysis and following rules that force mutexes to be taken in a certain order when a thread must take multiple mutexes simultaneously.

However, because the Event and Receive threads already hold mutexes when invoking user callbacks, and because the Connext DDS APIs that the user code can invoke may try to take other mutexes, deadlock may still result. Thus, to prevent user code to cause internal Connext DDS threads to deadlock, we have created a concept called Exclusive Areas (EA) that follow rules that prevent deadlock. The more EAs that exist in a system, the more concurrency is allowed through Connext DDS code. However, the more EAs that exist, the more restrictions on the Connext DDS APIs that are allowed to be invoked in Entity Listener callbacks.

The 7.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) on page 350 control how many EAs will be created by Connext DDS. For a more detailed discussion on EAs and the restrictions on the use of Connext DDS APIs within Entity Listener methods, please see 4.5 Exclusive Areas (EAs) on page 197.

### 21.5 Controlling CPU Core Affinity for RTI Threads

Two fields in the DDS_ThreadSettings_t structure (see 19.4.7 Thread Settings on page 882) are related to CPU core affinity: cpu_list and cpu_rotation.

**Note:** Although DDS_ThreadSettings_t is used in the Event, Database, ReceiverPool, and AsynchronousPublisher QoS policies, cpu_list and cpu_rotation are only relevant in the 9.5.6 RECEIVER_POOL QosPolicy (DDS Extension) on page 664.

While most thread-related QoS settings apply to a single thread, the ReceiverPool QoS policy’s thread-settings control every receive thread created. In this case, there are several schemes to map $M$ threads to $N$ processors; cpu_rotation controls which scheme is used.

The cpu_rotation determines how cpu_list affects processor affinity for thread-related QoS policies that apply to multiple threads. If cpu_list is empty, cpu_rotation is irrelevant since no affinity adjustment will
occur. Suppose instead that \texttt{cpu\_list} = \{0,1\} and that the middleware creates three receive threads: \{A, B, C\}. If \texttt{cpu\_rotation} is set to \texttt{CPU\_NO\_ROTATION}, threads A, B and C will have the same processor affinities (0-1), and the OS will control thread scheduling within this bound.

CPU affinities are commonly denoted with a bitmask, where set bits represent allowed processors to run on. This mask is printed in hex, so a CPU affinity of 0-1 can be represented by the mask 0x3.

If \texttt{cpu\_rotation} is \texttt{CPU\_RR\_ROTATION}, each thread will be assigned in round-robin fashion to one of the processors in \texttt{cpu\_list}; perhaps thread A to 0, B to 1, and C to 0. Note that the order in which internal middleware threads spawn is unspecified.

The \texttt{RTI Connext DDS Core Libraries Platform Notes} describe which architectures support this feature.

### 21.6 Configuring Thread Settings with XML

Table 21.1 XML Tags for ThreadSettings_t describes the XML tags that you can use to configure thread settings. For more information on thread settings, see:

- 19.4.7 Thread Settings on page 882
- The \texttt{RTI Connext DDS Core Libraries Platform Notes}
- The API Reference HTML documentation (select Modules, RTI Connext DDS API Reference, Infrastructure Module, QoS Policies, Extended QoS Support, Thread Settings)

#### Table 21.1 XML Tags for ThreadSettings_t

<table>
<thead>
<tr>
<th>Tags within &lt;thread&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;cpu_list&gt;</td>
<td>Each &lt;element&gt; specifies a processor on which the thread may run. Only applies to platforms that support controlling CPU core affinity (see 21.5 Controlling CPU Core Affinity for RTI Threads on the previous page and the \texttt{RTI Connext DDS Core Libraries Platform Notes}).</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| <cpu\_rotation>     | Determines how the CPUs in <cpu\_list> will be used by the thread. The value can be either:  
  - THREAD\_SETTINGS\_CPU\_NO\_ROTATION  
    The thread can run on any listed processor, as determined by OS scheduling.  
  - THREAD\_SETTINGS\_CPU\_RR\_ROTATION  
    The thread will be assigned a CPU from the list in round-robin order. Only applies to platforms that support controlling CPU core affinity (see the \texttt{RTI Connext DDS Core Libraries Platform Notes}). | 0 or 1 |
Table 21.1 XML Tags for ThreadSettings_t

<table>
<thead>
<tr>
<th>Tags within &lt;thread&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
</table>
| <mask>               | A collection of flags used to configure threads of execution. Not all of these options may be relevant for all operating systems. May include these bits:  
  - STDIO  
  - FLOATING_POINT  
  - REALTIME_PRIORITY  
  - PRIORITY_ENFORCE  
  It can also be set to a combination of the above bits by using the "or" symbol (|), such as STDIO|FLOATING_POINT.  
  Default: MASK_DEFAULT | 0 or 1 |
| <priority>           | Thread priority. The value can be specified as an unsigned integer or one of the following strings.  
  - THREAD_PRIORITY_DEFAULT  
  - THREAD_PRIORITY_HIGH  
  - THREAD_PRIORITY_ABOVE_NORMAL  
  - THREAD_PRIORITY_NORMAL  
  - THREAD_PRIORITY BELOW_NORMAL  
  - THREAD_PRIORITY_LOW  
  When using an unsigned integer, the allowed range is platform-dependent.  
  When thread priorities are configured using XML, the values are considered native priorities.  
  Example:  
  <thread>  
  <mask>STDIO|FLOATING_POINT</mask>  
  <priority>10</priority>  
  <stack_size>THREAD_STACK_SIZE_DEFAULT</stack_size>  
  </thread>  
  When the XML file is loaded using the Java API, the priority is a native priority, not a Java thread priority. | 0 or 1 |
| <stack_size>         | Thread stack size, specified as an unsigned integer or set to the string THREAD_STACK_SIZE_DEFAULT. The allowed range is platform-dependent. | 0 or 1 |

21.7 User-Managed Threads

In certain scenarios, you may want full control over the internal threads created by your Connext DDS applications. For instance, in memory-constrained systems, applications may want to manage the resources required by internal Connext DDS threads. Also, you may want to use a different thread technology than the one Connext DDS incorporates by default (i.e., pthread on POSIX platforms).
Connext DDS can create the internal threads from the application layer via the abstract factory pattern. You can provide a Connext DDS application with a ThreadFactory implementation that DomainParticipants will use to create and delete all the threads.

The ThreadFactory interface exposes operations for creating and deleting threads. These operations are called on demand as DomainParticipants require new threads or need to delete existing ones.

The same ThreadFactory instance can be used by multiple DomainParticipants. To select which ThreadFactory to use, use the set_thread_factory() operation in the DomainParticipantFactory:

```cpp
MyThreadFactory myThreadFactory; // Implements DDSThreadFactory
retcode = DDSTheParticipantFactory->set_thread_factory(&myThreadFactory);
```

Then you can create DomainParticipants using any of the available APIs (i.e. create_participant(), create_participant_from_config(), etc). A DomainParticipant will use the ThreadFactory object that is set in the DomainParticipantFactory at the time it is created and throughout its entire lifecycle. If a new ThreadFactory is set, existing DomainParticipants will not be affected; they will still use the same ThreadFactory with which they were created.

This feature is only available for the C/C++ APIs. For further information, please see the API Reference HTML documentation.

### 21.8 Unregistering Threads

If the logic of your application requires spawning new threads, and in those threads you are calling a Connext DDS API such as write(), you may notice a memory growth.

To ensure that all the resources allocated in that thread are correctly released, remember to call unregister_thread() right before exiting the thread.

Here is what the full API looks like:

```cpp
DDS_ReturnCode_t DDS_DomainParticipantFactory_unregister_thread(DDS_DomainParticipantFactory *self);
```

For more information, search for DDS_DomainParticipantFactory_unregister_thread in the API Reference HTML documentation.

### 21.9 Identifying Threads Used by Connext DDS

Connext DDS uses multiple internal threads for sending and receiving data, maintaining internal state, and calling user code when events occur. Further details regarding Connext DDS’s threading model can be found in Chapter 21 Connext DDS Threading Model on page 908. This section explains how these threads can be identified in your system.
21.9.1 Checking Thread Names at the OS Level

On some systems, it is possible to check the internal name of RTI threads directly at the operating system level. Threads created by Connext DDS will have RTI-specific thread names, unless otherwise stated in the RTI Connext DDS Core Libraries Platform Notes, which lists architectures that do not support setting thread names.

In general, thread names follow this pattern:

\[ r<\text{Module}>[<\text{Participant identifier}>][<\text{Thread index}>][<\text{Transport name}>]<\text{Task type}> \]

Where:

- The maximum length for a thread name is 16, including the '\0'.
- \( r \) indicates this is a thread from RTI.
- The second and third characters identify the \(<\text{Module}>\):

<table>
<thead>
<tr>
<th>Module</th>
<th>Thread Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core</td>
<td>Co</td>
</tr>
<tr>
<td>Transport</td>
<td>Tr</td>
</tr>
<tr>
<td>Security</td>
<td>Se</td>
</tr>
<tr>
<td>Distributed Logger</td>
<td>DI</td>
</tr>
<tr>
<td>Persistence Service</td>
<td>Ps</td>
</tr>
<tr>
<td>Database Integration Service</td>
<td>Ds</td>
</tr>
<tr>
<td>Web Integration Service</td>
<td>Ws</td>
</tr>
<tr>
<td>Monitor</td>
<td>Mo</td>
</tr>
<tr>
<td>Recording Service</td>
<td>Re</td>
</tr>
<tr>
<td>Routing Service</td>
<td>Rs</td>
</tr>
</tbody>
</table>

- \(<\text{Participant identifier}>\) is represented with five characters, as follows:
  - If \text{participant\_name} is set: the participant identifier will be the first three characters and the last two characters of the \text{participant\_name}.
  - If \text{participant\_name} is not set: the identifier is computed as \text{domain\_id} (three characters), \text{participant\_id} (two characters).
21.9.1 Checking Thread Names at the OS Level

- If `participant_name` is not set and the `participant_id` is set to -1 (default value): the participant identifier is computed as the last five digits of the `rtps_instance_id` in the participant GUID.

- `<Thread index>` - index used to distinguish among threads with the same name.

For example, there are several instantiations of the receive thread; the thread index is used to differentiate them:

```
rCo32265##00Rcv
rCo32265##01Rcv
rCo32265##02Rcv
rCo32265##03Rcv
rCo32265##04Rcv
```

- `<Transport name>` is represented with four characters:

<table>
<thead>
<tr>
<th>Transport Name</th>
<th>Thread Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transmission Control Protocol version 4 (TCPv4)</td>
<td>TCP4</td>
</tr>
<tr>
<td>Datagram Transport Layer Security (DTLS)</td>
<td>DTLS</td>
</tr>
<tr>
<td>Transport Layer Security (TLS)</td>
<td>TLS</td>
</tr>
<tr>
<td>Wide Area Network (WAN)</td>
<td>WAN</td>
</tr>
<tr>
<td>User Datagram Protocol version 4 (UDPv4)</td>
<td>UDP4</td>
</tr>
<tr>
<td>User Datagram Protocol version 6 (UDPv6)</td>
<td>UDP6</td>
</tr>
</tbody>
</table>

- `<taskType>` - the type of thread is represented with three characters:

<table>
<thead>
<tr>
<th>Transport Name</th>
<th>Thread Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event</td>
<td>Evt</td>
</tr>
<tr>
<td>Receive</td>
<td>Rcv</td>
</tr>
<tr>
<td>Database</td>
<td>Dtb</td>
</tr>
<tr>
<td>Asynchronous waitSet</td>
<td>AWs</td>
</tr>
<tr>
<td>Dispatcher</td>
<td>Dsp</td>
</tr>
</tbody>
</table>
The details on checking the thread names depend on the operating system. The following is an example output from a publisher application running on VxWorks 6.9.4:

```bash
-> taskSpawn "test", 255, <floating_point_option>, 150000, publisher_main, 1, 100
value = 83748528 = 0x4fde6b0
-> i
NAME     ENTRY   TID  PRI   STATUS  PC  SP   ERRNO  CPU #
--------- ------- ----- ----- ------- ------ ------ -------- ------
[...]
   rCoHelnt##> RTIOsapiThr> 444a010 71 PEND  37b218 53f7c00 0 -
   rCoHelnt##> RTIOsapiThr> 540b2b0 71 PEND  37b218 542ac00 0 -
   rCoHelnt##> RTIOsapiThr> 543e080 71 PEND  37b218 545dc90 0 -
   rCoHelnt##> RTIOsapiThr> 543ea78 71 PEND  37b218 549c00 0 -
   Test    REDATester_ 456c010 100 STOP 2cf594 501cc94 0 -
   rCoHelnt##> RTIOsapiThr> 44d4358 110 PEND+T 37b218 735fda0 3d0010 -
   rCoHelnt##> RTIOsapiThr> 44dea68 120 PEND+T 37b218 730fe04 0 -
   rTr HelntUD> RTIOsapiThr> 53bbcb0 120 PEND+T 37b218 53c4e2c 0 -
   tZync7Task 2c8e9c 43f0228 240 DELAY 384288 4df5fa4 0 -
   miiBusMoni> 2c9974 464bb60 252 DELAY 384288 4654fb8 0 -
   test     publisher_m 4fde6b0 255 DELAY 384288 721ee8 3d0010 -
   tIdleTask0 idleTaskEnt> 43d5418 287 READY 37a918 43d53ec 0 -
   tIdleTask1 idleTaskEnt> 43d9670 287 READY 37a918 43d9644 0 -
value = 0 = 0x0
-> ti 0x444a010
NAME     ENTRY   TID  PRI   STATUS  PC  SP   ERRNO  DELAY
--------- ------- ----- ----- ------- ------ ------ -------- ------
rCoHelnt##> RTIOsapiThr> 444a010 71 PEND 37b218 53f7c00 0 0
full task name : rCoHelnt##00Rcv
```
21.9.1 Checking Thread Names at the OS Level

```
<table>
<thead>
<tr>
<th>task entry</th>
<th>RTIOSapiThreadChild_onSpawned</th>
</tr>
</thead>
<tbody>
<tr>
<td>task affinity</td>
<td>0x00000000</td>
</tr>
<tr>
<td>[...]</td>
<td></td>
</tr>
</tbody>
</table>
```

Where `<floating_point_option>` is a numeric value that varies depending on the hardware. See Enabling Floating Point Coprocessor in Kernel Tasks, in the RTI Connext DDS Core Libraries Platform Notes.

In this example, the `i` command in VxWorks retrieves information about the running threads. The `>` at the end of the name (`rCoHelnt##`) indicates that the full thread name could not be displayed, because it exceeds 10 characters. You can use the `ti` command in VxWorks (shown above), followed by the thread ID (TID), to retrieve information about a specific thread, including its full name (in this case, `rCoHelnt##00Rcv`).

The following is an example from running a subscriber on a Linux machine:

```
$ ./objs/x64Linux3gcc5.4.0/HelloWorld_subscriber
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
[...]
$ ps -eT | grep rC
22966 22967 pts/19   00:00:00 rCo32265####Dtb
22966 22968 pts/19   00:00:00 rCo32265####Evt
22966 22970 pts/19   00:00:00 rCo32265##00Rcv
22966 22971 pts/19   00:00:00 rCo32265##01Rcv
22966 22972 pts/19   00:00:00 rCo32265##02Rcv
22966 22973 pts/19   00:00:00 rCo32265##03Rcv
22966 22974 pts/19   00:00:00 rCo32265##04Rcv
```

**Note:** For transport threads, you have the option of setting your own thread name prefix, which substitutes the first three components (`r<Module>[<Participant identifier>]`) of the thread name with your own prefix. Setting your own thread name prefix allows you to add extra information to the transport thread, such as your own identifier for the threads or the *Topic* used. You can optionally set this prefix using the `thread_name_prefix` field in the transport (for example, in the 44.7 TCP/TLS Transport Properties on page 1158).

Table 21.5 Example Thread Names shows names for the majority of threads created by *Connext DDS*.
### Table 21.5 Example Thread Names

<table>
<thead>
<tr>
<th>Thread Information</th>
<th>Name</th>
<th>Fields</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receive thread</td>
<td>rCo%5s##%02dRcv</td>
<td>Participant identifier, thread index</td>
<td>rCo11122##33Rcv</td>
</tr>
<tr>
<td>Asynchronous waitset thread</td>
<td>rCo%5s##%02dAWs</td>
<td>Participant identifier, thread index</td>
<td>rCo11122##33AWs</td>
</tr>
<tr>
<td>Database thread</td>
<td>rCo%5s###Db</td>
<td>Participant identifier</td>
<td>rCo11122####Db</td>
</tr>
<tr>
<td>Dispatcher (i.e., asynchronous publishing) thread</td>
<td>rCo%5s##%02dDsp</td>
<td>Participant identifier, thread index</td>
<td>rCo11122##33Dsp</td>
</tr>
<tr>
<td>Asynchronous batch flushing thread</td>
<td>rCo%5s###ABF</td>
<td>Participant identifier</td>
<td>rCo11122####ABF</td>
</tr>
<tr>
<td>Topic query publication thread</td>
<td>rCo%5s###TQP</td>
<td>Participant identifier</td>
<td>rCo11122####TQP</td>
</tr>
<tr>
<td>Event thread</td>
<td>rCo%5s###Evt</td>
<td>Participant identifier</td>
<td>rCo11122####Evt</td>
</tr>
<tr>
<td>DNS tracker thread</td>
<td>rCo%5s###DNS</td>
<td>Participant identifier</td>
<td>rCo11122####DNS</td>
</tr>
<tr>
<td>Distributed logger writer thread</td>
<td>rDl########################################################Wri</td>
<td>Participant identifier</td>
<td>rDl########################################################Wri</td>
</tr>
<tr>
<td>Secure distributed logger thread</td>
<td>rSe%5s###Log</td>
<td>Participant identifier</td>
<td>rSe11122####Log</td>
</tr>
<tr>
<td>TCP control thread</td>
<td>rTr%5s%04sCtr</td>
<td>Participant identification, transportName (TCP4)</td>
<td>rTr11122TCP4Ctr</td>
</tr>
<tr>
<td>TCP event thread</td>
<td>rTr%5s%04sEvt</td>
<td>Participant identification, transportName (TCP4)</td>
<td>rTr11122TCP4Evt</td>
</tr>
<tr>
<td>DTLS event thread</td>
<td>rTr%5s%04sEvt</td>
<td>Participant identification, transportName (DTLS)</td>
<td>rTr11122DTLSEvt</td>
</tr>
<tr>
<td>TLS receive thread</td>
<td>rTr%5s%04sRcv</td>
<td>Participant identification, transportName (TLS)</td>
<td>rTr11122#TLSRcv</td>
</tr>
<tr>
<td>WAN receive thread</td>
<td>rTr%5s%04sRcv</td>
<td>Participant identification, transportName (WAN)</td>
<td>rTr11122#WANRcv</td>
</tr>
<tr>
<td>WAN server thread</td>
<td>rTr%5s%04sSvr</td>
<td>Participant identification, transportName (WAN)</td>
<td>rTr11122#WANCtr</td>
</tr>
<tr>
<td>Interface tracking thread</td>
<td>rTr%5s%04sITr</td>
<td>Participant identification, transportName (UDP4, UDP6, TCP4)</td>
<td>rTr11122UDP4ITr</td>
</tr>
<tr>
<td>Persistence Service receive admin command request thread</td>
<td>rPs%03d####RAC</td>
<td>domainId</td>
<td>rPs1111####RAC</td>
</tr>
</tbody>
</table>
## 21.9.1 Checking Thread Names at the OS Level

<table>
<thead>
<tr>
<th>Thread Information</th>
<th>Name</th>
<th>Fields</th>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistence Service discovery thread</td>
<td>rPs%09sDis</td>
<td>Application name</td>
<td>rPsTestPersistence</td>
</tr>
<tr>
<td>Persistence Service reception thread (topic)</td>
<td>rPs%07s%02dRcv</td>
<td>topic name, thread index</td>
<td>rPsHello##33Rea</td>
</tr>
<tr>
<td>Persistence Service publication thread</td>
<td>rPs%07s%02dPub</td>
<td>topic name, thread index</td>
<td>rPsHello##33Pub</td>
</tr>
<tr>
<td>Persistence Service reception thread (Top-icSet)</td>
<td>rPsTopic##%02dRcv</td>
<td>thread index</td>
<td>rPsHello##33Rea</td>
</tr>
<tr>
<td>Persistence Service event thread</td>
<td>rPs############Evt</td>
<td></td>
<td>rPsHello##33Rea</td>
</tr>
<tr>
<td>Recording Service timer thread</td>
<td>rRe#############Tim</td>
<td></td>
<td>rRe#############Tim</td>
</tr>
<tr>
<td>Monitor event thread</td>
<td>rMo%5s####Evt</td>
<td>Participant identifier</td>
<td>rREHelloWorlPub</td>
</tr>
<tr>
<td>Routing Service polling timer thread</td>
<td>rRs################Tim</td>
<td></td>
<td>rRs################Tim</td>
</tr>
<tr>
<td>Routing Service filter tracker event thread</td>
<td>rRsFilterTr##Evt</td>
<td></td>
<td>rRsFilterTr##Evt</td>
</tr>
<tr>
<td>Routing Service monitor statistics event thread</td>
<td>rRsMoSta#######Evt</td>
<td></td>
<td>rRsMoSta#######Evt</td>
</tr>
<tr>
<td>Routing Service monitor publication event thread</td>
<td>rRsMoPub######Evt</td>
<td></td>
<td>rRsMoPub######Evt</td>
</tr>
<tr>
<td>Routing Service discovery event thread</td>
<td>rRsDisc######Evt</td>
<td></td>
<td>rRsDisc######Evt</td>
</tr>
<tr>
<td>Routing Service asynchronous admin thread</td>
<td>rRsAdmin##%02dAWs</td>
<td>thread index</td>
<td>rRsAdmin##%33AWs</td>
</tr>
<tr>
<td>Routing Service asynchronous discovery thread</td>
<td>rRsDisc##%02dAWs</td>
<td>thread index</td>
<td>rRsDisc##33dAWs</td>
</tr>
<tr>
<td>Database Integrated Service discovery thread</td>
<td>rDs###########Dis</td>
<td></td>
<td>rDs###########Dis</td>
</tr>
<tr>
<td>Database Integrated Service connection thread</td>
<td>rDs%.9sCon</td>
<td>Database name</td>
<td>rDsTestsCon</td>
</tr>
<tr>
<td>Database Integrated Service refresh thread</td>
<td>rDs%.9sRef</td>
<td>Database name</td>
<td>rDsTestsRef</td>
</tr>
<tr>
<td>Database Integrated Service finalization Library thread</td>
<td>rDsFinalizeLib#</td>
<td></td>
<td>rDsFinalizeLib#</td>
</tr>
<tr>
<td>Database Integrated Service event manager thread</td>
<td>rDsManager####Evt</td>
<td></td>
<td>rDsManager####Evt</td>
</tr>
<tr>
<td>Web Integrated Service access control list database thread</td>
<td>rWsACL#######Dtb</td>
<td></td>
<td>rWsACL#######Dtb</td>
</tr>
</tbody>
</table>
21.9.2 Checking Thread Names from the Call Stack

Thread names are only available in a subset of architectures. See the RTI Connext DDS Core Libraries Platform Notes for which architectures support checking thread names at the OS level. This section lists the correspondence between Connext DDS threads and the functions they run. You can use this information to identify Connext DDS threads from the call stack, independently of your architecture. If you are using VxWorks or Integrity, see 21.9.1 Checking Thread Names at the OS Level on page 917.

This is the correspondence between threads and the functions they run:

- **Database Thread**: RTIEventActiveDatabaseThread_loop()
- **(Main) Event Thread**: RTIEventActiveGeneratorThread_loop(). Note that this function is generic to all the event threads. That is, all of the event threads run RTIEventActiveGeneratorThread_loop(), which detects and handles events. For this reason, it can be difficult to distinguish the Main Event Thread from other event threads (such as the Topic Query Publication Event Thread); however, to better make this distinction, you can check whether some (sub)functions are called (for example, the subfunctions related to the Asynchronous Batch Flushing Event Thread and Topic Query Publication Event Thread below).
- **Receive Thread**: COMMENDActiveFacadeReceiver_loop(), which calls to a different function depending on what transport is being used to get the (meta)data:
  - **Shared Memory**: NDDS_Transport_Shmem_receive_rEA()
  - **UDP**: NDDS_Transport_UDP_receive_rEA()
  - **TCP**: NDDS_Transport_TCP_receive_rEA()
- **Interface Tracking Thread**: RTIOsapiInterfaceTracker()
- **Transport-Specific Threads**:
  - **TCP Control Thread**: NDDS_Transport_TCPv4_Plugin_threadLoop()
  - **TCP Event Thread**: RTIEventActiveGeneratorThread_loop() and NDDS_Transport_TCPv4_Plugin_clientOn<event_name>()
- **Asynchronous Publishing Thread**: RTIEventJobDispatcherThread_spawnedFnc()
- **Asynchronous Batch Flushing Event Thread**: RTIEventActiveGeneratorThread_loop() and PRESPsWriter_onFlushBatch()
- **Topic Query Publication Event Thread**: RTIEventActiveGeneratorThread_loop() and PRESPsService_onWriterServiceDispatchActiveTopicQueriesEvent()

For example, if you are on GNU/Linux, you can run the following command on gdb to get the call stack:
The same information can be seen with Visual Studio. To see this information in Visual Studio, select Debug > Windows > Threads, then do Ctrl+D, T. You will need to add a breakpoint and start the application in debug mode.

### 21.9.3 Checking Thread Names Using the Worker’s Name

Connext DDS uses the concept of a worker as an abstraction for threads. Workers are RTI-specific entities used internally to manage critical sections and to provide access to thread-specific storage. Most of the threads created by Connext DDS have an associated worker. In addition, user threads calling certain APIs from Connext DDS will have a worker associated with them. Workers are given a name when they are created. If you have the proper debug symbols, you can use the worker’s name to identify the thread (on a debugger, for instance).

To check the workers’ names, first locate these workers in the threads. You can do that by selecting a thread and printing its full backtrace. Another option is moving up and down through the frames on the thread’s stack. The worker will be either a local variable or the last argument to one of the RTI functions. Here is an example using gdb on GNU/Linux to identify a thread with the method just described:

```plaintext
(gdb) info thread
    Id Target Id   Frame
    * 1  Thread 0x7fffffffe700 (LWP 6801) "HelloWorld_publ" __clock_nanosleep (clock_id=<optimized out>, flags=0, req=0x7fffffffcb20, rem=0x7fffffffcb30) at ../sysdeps/unix/sysv/linux/clock_nanosleep.c:48
    2  Thread 0x7ffffff6ec1700 (LWP 6805) "HelloWorld_publ" pthread_cond_timedwait@@GLIBC_2.3.2 () at ../sysdeps/unix/sysv/linux/x86_64/pthread_cond_timedwait.S:225
    3  Thread 0x7ffffff66c0700 (LWP 6806) "HelloWorld_publ" pthread_cond_timedwait@@GLIBC_2.3.2 () at ../sysdeps/unix/sysv/linux/x86_64/pthread_cond_timedwait.S:225

(gdb) thread 2
[Switching to thread 2 (Thread 0x7ffffff6ec1700 (LWP 6805))]
```

```plaintext
(gdb) backtrace full
 [...] #3 0x0000000000c6095b in RTIEventActiveDatabaseThread_loop (param=0x13fc8c0) at ActiveDatabase.c:156
    timeStr = 0x7ffffff6ec0dc0 "(0000003d,00000000)"
    t = 0x13fc8c0
    canBeDeleted = 0
    timeBuf = "{0000003d,00000000}"
    workerName = 0x1351cb0 "rDtb2081a9101"
    METHOD_NAME = 0x100a0d0 "RTIEventActiveDatabaseThread_loop"
 [...]```

As you can see in the example, workers follow the same naming convention as threads (in some cases, a shortened version of it). Workers associated with user threads use the following convention: U<threadId>, where:
21.9.3 Checking Thread Names Using the Worker’s Name

- **U** - indicator that this is a User Thread
- `<threadId>` - ID given to the thread by the OS
# Chapter 22 DDS Sample and Instance Memory Management

This chapter describes how Connext DDS manages the memory for the DDS data samples that are sent by DataWriters and received by DataReaders.

## 22.1 DDS Sample Memory Management for DataWriters

To configure DDS sample memory management on the writer side, use the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439. Table 22.1 DDS Sample Memory Management Properties for DataWriters lists the supported memory-management properties for DataWriters. This section applies to DataWriters that use IDL-generated type-plugins as well as DynamicDataWriters.

### Table 22.1 DDS Sample Memory Management Properties for DataWriters

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size | If the serialized size of the DDS sample is <= pool_buffer_max_size: The buffer is obtained from a pre-allocated pool and released when the DataWriter is deleted.  
If the serialized size of the DDS sample is > pool_buffer_max_size: The buffer is dynamically allocated from the heap and returned to the heap when the DDS sample is removed from the DataWriter’s queue. The size of the buffer allocated from the heap is the sample serialized size.  
Default: -1 (UNLIMITED). All DDS sample buffers are obtained from the pre-allocated pool; the buffer size is the maximum serialized size of the DDS samples, as returned by the type plugin get_serialized_sample_max_size() operation.  
Notes:  
- If you use unbounded sequences or strings, then you should set pool_buffer_max_size to a finite value. See 3.1.1 Sequences on page 30.  
- The pool_buffer_max_size also controls the memory allocation for the serialized key buffer that is stored with every instance. See 22.3 Instance Memory Management for DataWriters on page 940.  

See 22.1.1 Memory Management without Batching on the next page.
### Table 22.1 DDS Sample Memory Management Properties for DataWriters

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.data_writer.history.memory_manager.java_stream.min_size</td>
<td>Only supported when using the Java API. Defines the minimum size of the buffer that will be used to serialize DDS samples. When a DataWriter is created, the Java layer will allocate a buffer of this size and associate it with the DataWriter. Default: -1 (UNLIMITED). This is a sentinel that refers to the maximum serialized size of a DDS sample, as returned by the type plugin <code>get_serialized_sample_max_size()</code> operation. See 22.1.3 Writer-Side Memory Management when Using Java on page 931.</td>
</tr>
<tr>
<td>dds.data_writer.history.memory_manager.java_stream.trim_to_size</td>
<td>Only supported when using the Java API. A boolean value that controls the growth of the serialization buffer. If set to 0 (default): The buffer will not be reallocated unless the serialized size of a new DDS sample is greater than the current buffer size. If set to 1: The buffer will be reallocated with each new DDS sample to a smaller size in order to just fit the DDS sample serialized size. The new size cannot be smaller than <code>min_size</code>. See 22.1.3 Writer-Side Memory Management when Using Java on page 931.</td>
</tr>
</tbody>
</table>

### 22.1.1 Memory Management without Batching

When the `write()` operation is called on a DataWriter that does not have batching enabled, the DataWriter serializes (marshals) the input DDS sample and stores it in the DataWriter’s queue (see Figure 22.1: DataWriter Actions when Batching is Disabled on the next page). The size of this queue is limited by `initial_samples/max_samples` in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.
Each DDS sample in the queue has an associated serialization buffer in which the DataWriter will serialize the DDS sample. This buffer is either obtained from a pre-allocated pool (if the serialized size of the DDS sample is \( \leq \) `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size`) or the buffer is dynamically allocated from the heap (if the serialized size of the DDS sample is \( > \) `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size`). The size of the buffer allocated on the heap is the sample serialized size. See Table 22.1 DDS Sample Memory Management Properties for DataWriters.

The default value of `pool_buffer_max_size` is -1 (UNLIMITED). In this case, all the DDS samples come from the pre-allocated pool and the size of the buffers is the maximum serialized size of the DDS samples as returned by the type plugin `get_serialized_sample_max_size()` operation. The default value is optimum for real-time applications where determinism and predictability is a must. The trade-off is higher memory usage, especially in cases where the maximum serialized size of a DDS sample is large. If the maximum
serialized size of a DDS sample is large, but bounded, the value can be set to a finite value to save memory. If the maximum serialized size is unbounded, then the value must be set to a finite value in order to avoid running out of system memory while allocating the sample pools.

Connext DDS cannot send arbitrarily large samples. For details on serialization limits see 3.10 Data Sample Serialization Limits on page 164.

22.1.2 Memory Management with Batching

When the write() operation is called on a DataWriter for which batching is enabled (see 7.5.2 BATCH QosPolicy (DDS Extension) on page 374), the DataWriter serializes (marshals) the input DDS sample into the current batch buffer (see Figure 22.2: DataWriter Actions when Batching is Enabled on the next page). When the batch is flushed, it is stored in the DataWriter’s queue along with its DDS samples. The DataWriter queue can be sized based on:

- The number of DDS samples, using initial_samples/max_samples (both set in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451)
- The number of batches, using initial_batches/max_batches (both set in the 7.5.6 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 401)
- Or a combination of max_samples and max_batches
Figure 22.2: DataWriter Actions when Batching is Enabled

Initial samples:
- resource_limits.initial_samples

Maximum samples:
- resource_limits.max_samples

Initial batches:
- writer_resource_limits.initial_batches

Maximum batches:
- writer_resource_limits.max_batches

DataWriter’s Queue
22.1.3 Writer-Side Memory Management when Using Java

When batching is enabled, the memory associated with the batch buffers always comes from a pre-allocated pool. The size of the buffers is determined by the QoS values max_samples and max_data_bytes (both set in the 7.5.2 BATCH QosPolicy (DDS Extension) on page 374) as follows:

- If max_data_bytes is a finite value, the size of the buffer is the maximum of this value and the maximum serialized size of a DDS sample (max_sample_serialized_size) as returned by the type-plugin get_serialized_sample_max_size(), since that batch must contain at least one DDS sample.
- Otherwise, the size of the buffer is calculated by (batch.max_samples * max_sample_serialized_size).

Notice that for variable-size DDS samples (for example, DDS samples containing sequences) it is good practice to size the buffer based on max_data_bytes, since this leads to more efficient memory usage.

Note: The value of the property dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size is ignored by DataWriters with batching enabled.

22.1.3 Writer-Side Memory Management when Using Java

When the Java API is used, Connext DDS allocates a Java buffer per DataWriter; this buffer is used to serialize the Java DDS samples published by the DataWriters. After a DDS sample is serialized into a Java buffer, the result is copied into the underlying native buffer described in 22.1.1 Memory Management without Batching on page 927 and 22.1.2 Memory Management with Batching on page 929.

You can use the following two DataWriter properties to control memory allocation for the Java buffers that are used for serialization (see Table 22.1 DDS Sample Memory Management Properties for DataWriters):

- dds.data_writer.history.memory_manager.java_stream.min_size
- dds.data_writer.history.memory_manager.java_stream.trim_to_size

22.1.4 Writer-Side Memory Management when Working with Large Data

Large DDS samples are DDS samples with a large maximum size relative to the memory available to the application. Notice the use of the word maximum, as opposed to actual size.

As described in 22.1.1 Memory Management without Batching on page 927, by default, the middleware preallocates the DDS samples in the DataWriter queue to their maximum serialized size. This may lead to high memory-usage in DataWriters where the maximum serialized size of a DDS sample is large.

For example, let’s consider a video conferencing application:
22.1.4 Writer-Side Memory Management when Working with Large Data

The above IDL definition can be used to work with video streams.

Each frame is transmitted as a sequence of octets with a maximum size of 1 MB. In this example, the video stream has two types of frames: I-frames (also called key frames) and P-frames (also called delta frames). I-frames represent full images and do not require information about the preceding frames in order to be decoded. P-frames require information about the preceding frames in order to be decoded.

A video stream consists of a sequence of frames in which I-frames are followed by multiple P-frames. The number of P-frames between I-frames affects the video quality since, in a non-reliable configuration, losing a P-frame will degrade the image quality until the next I-frame is received.

For our use case, let’s assume that I-frames may require 1 MB, while P-frames require less than 32 KB. Also, there are 20 times more P-frames than I-frames.

Although the actual size of the frames sent by the Connext DDS application is usually significantly smaller than 1 MB since they are P-frames, the default memory management will use 1 MB per frame in the DataWriter queue. If resource_limits.max_samples is 256, the DataWriter may end up allocating 256 MB.

Using some domain-specific knowledge, such as the fact that most of the P-frames have a size smaller than 32 KB, we can optimize memory usage in the DataWriter’s queue while still maintaining determinism and predictability for the majority of the frames sent on the wire.

The following XML file shows how to optimize the memory usage for the previous example (rather than focusing on efficient usage of the available network bandwidth).

```xml
<?$xml version="1.0"?>
<!-- XML QoS Profile for large data -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- QoS Library containing the QoS profile used for large data -->
  <qos_library name="ReliableLargeDataLibrary">
    <!-- QoS profile to optimize memory usage in DataWriters sending large images -->
    <qos_profile name="ReliableLargeDataProfile"
      is_default_qos="true">
      <!-- QoS used to configure the DataWriter -->
      <datawriter_qos>
        <resource_limits>
          <max_samples>32</max_samples>
          <!-- No need to pre-allocate 32 images unless needed -->
          <initial_samples>1</initial_samples>
        </resource_limits>
      </datawriter_qos>
    </qos_profile>
  </qos_library>
</dds>
```

932
equal to 33 KB the serialization buffer is obtained from a pre-allocated pool. For sizes greater than 33 KB, the DataWriter will use dynamic memory allocation.

```xml
<element>
  <name>
  dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size
  </name>
  <value>33792</value>
</element>

<!-- Java will use a 33 KB buffer to serialize all frames with a size smaller than or equal to 33 KB. When an I-frame is published, Java will reallocate the serialization buffer to match the serialized size of the new frame. -->

```xml
<element>
  <name>
  dds.data_writer.history.memory_manager.java_stream.min_size
  </name>
  <value>33792</value>
</element>
```

Working with large data DDS samples will likely require throttling the network traffic generated by single DDS samples. For additional information on shaping network traffic, see 7.6 FlowControllers (DDS Extension) on page 470.

### 22.2 DDS Sample Memory Management for DataReaders

The DDS data samples received by a DataReader are deserialized (demarshaled) and stored in the DataReader’s queue (see Figure 22.3: Adding DDS Samples to DataReader’s Queue on the next page). The size of this queue is limited by initial_samples/max_samples in the 7.5.22 RESOURCE_LIMITS QosPolicy on page 451.
22.2.1 Memory Management for DataReaders Using Generated Type-Plugins

Figure 22.3: Adding DDS Samples to DataReader’s Queue below shows how DDS samples are processed and added to the DataReader’s queue.

Figure 22.3: Adding DDS Samples to DataReader’s Queue

The RTPS DATA DDS samples received by a DataReader can be either batch DDS samples or individual DDS samples. The DataReader queue does not store batches. Therefore, each one of the DDS samples within a batch will be deserialized and processed individually.
When the `DataReader` processes a new sample, it will deserialize it into a sample obtained from a pre-allocated pool. By default, to provide predictability and determinism, the sample obtained from the pool is allocated to its maximum size. For example, with the following IDL type, each sample in the `DataReader` queue will consume 1 MB, even if the actual size is smaller.

```c
struct VideoFrame {
    boolean keyFrame;
    sequence<octet,1024000> data;
};
```

In the above example, it is possible to reduce the memory consumption by declaring the data sequence as unbounded and by generating code for the type with the command-line option `-unboundedSupport`. In this case, the middleware will not preallocate 1 MB for the data member. Instead, the generated code will deserialize incoming samples by dynamically allocating and deallocating memory to accommodate the actual size of the data sequence.

### 22.2.2 Reader-Side Memory Management when Using Java

When the Java API is used with `DataReaders` using generated type-plugins, `Connext DDS` allocates a Java buffer per `DataReader`; this buffer is used to copy the native serialized data, so that the received DDS samples can be deserialized into the Java objects obtained from the DDS sample pool in Figure 22.3: Adding DDS Samples to `DataReader`’s Queue on the previous page.

You can use the `DataReader` properties in Table 22.2 DDS Sample Memory Management Properties for `DataReaders` when Using Java API to control memory allocation for the Java buffer used for deserialization:

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.data_reader.</td>
<td>Only supported when using the Java API.</td>
</tr>
<tr>
<td>history.memory_manager</td>
<td></td>
</tr>
<tr>
<td>java_stream.min_size</td>
<td>Only supported when using the Java API.</td>
</tr>
<tr>
<td></td>
<td>Defines the minimum size of the buffer used for the serialized data.</td>
</tr>
<tr>
<td></td>
<td>When a <code>DataReader</code> is created, the Java layer will allocate a buffer of this size and associate it with the <code>DataReader</code>.</td>
</tr>
<tr>
<td></td>
<td>Default: -1 (UNLIMITED) This is a sentinel to refer to the maximum serialized size of a DDS sample, as returned by the type plugin method <code>get_serialized_sample_max_size()</code>.</td>
</tr>
<tr>
<td>dds.data_reader.</td>
<td>Only supported when using the Java API.</td>
</tr>
<tr>
<td>history.memory_manager</td>
<td></td>
</tr>
<tr>
<td>java_stream.trim_to_size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A Boolean value that controls the growth of the deserialization buffer.</td>
</tr>
<tr>
<td></td>
<td>If set to 0 (the default), the buffer will not be re-allocated unless the serialized size of a new DDS sample is greater than the current buffer size.</td>
</tr>
<tr>
<td></td>
<td>If set to 1, the buffer will be re-allocated with each new DDS sample in order to just fit the DDS sample serialized size. The new size cannot be smaller than <code>min_size</code>.</td>
</tr>
</tbody>
</table>
22.2.3 Memory Management for DynamicData DataReaders

Unlike DataReaders that use generated type-plugin code, DynamicData DataReaders provide configuration mechanisms to control the memory usage for use cases involving large data DDS samples. It is not required to set any of the following properties in order to support unbounded types in your application. The default behavior for a DynamicData DataReader is that samples are allocated to the minimum deserialized size and can grow to any size required to store incoming samples.

A DDS DynamicData sample stored in the DataReader’s queue has an associated underlying buffer that contains the DynamicData-specific representation of the DDS sample. The buffer is allocated according to the configuration provided in the data and serialization members of the DynamicDataTypeProperty_t used to create the DynamicDataTypeSupport (see 3.8 Interacting Dynamically with User Data Types on page 154).

```c
struct DDS_DynamicDataTypeProperty_t {
    DDS_DynamicDataProperty_t data;
    DDS_DynamicDataTypeSerializationProperty_t serialization;
};

struct DDS_DynamicDataTypeSerializationProperty_t {
    DDS_Boolean trim_to_size;
}

struct DDS_DynamicDataProperty_t {
    DDS_Long buffer_initial_size;
    DDS_Long buffer_max_size;
}
```

Table 22.3 struct DDS_DynamicDataTypeSerializationProperty_t below describes the member of DDS_DynamicDataTypeSerializationProperty_t.

Table 22.3 struct DDS_DynamicDataTypeSerializationProperty_t

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>trim_to_size</td>
<td>Controls the growth of the serialization buffer in a DynamicData object.</td>
</tr>
<tr>
<td></td>
<td>This property only applies to DynamicData samples that are obtained from</td>
</tr>
<tr>
<td></td>
<td>the sample pool that is created by each DynamicData DataReader.</td>
</tr>
<tr>
<td></td>
<td>If set to 0 (default): The buffer will not be reallocated unless the</td>
</tr>
<tr>
<td></td>
<td>deserialized size of the incoming DDS sample is greater than the current</td>
</tr>
<tr>
<td></td>
<td>buffer size.</td>
</tr>
<tr>
<td></td>
<td>If set to 1: The buffer of a DynamicData object obtained from the DDS</td>
</tr>
<tr>
<td></td>
<td>sample pool will be re-allocated for each sample to just fit the size of</td>
</tr>
<tr>
<td></td>
<td>the deserialized data of the incoming sample. The newly allocated size will</td>
</tr>
<tr>
<td></td>
<td>not be smaller than max,min_deserialized_size,buffer_initial_size.</td>
</tr>
</tbody>
</table>

The following table describes the members of DDS_DynamicDataProperty_t.
Table 22.4 struct DDS_DynamicDataProperty_t

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer_initial_size</td>
<td>The initial amount of memory used by the underlying DynamicData buffer, in bytes. This property is used to configure the DynamicData objects that are created stand-alone as well as the DynamicData samples that are obtained from the sample pool that is created by each DynamicData DataReader. If set to 0 (default): The initial buffer size will be set to the minimum amount of space required to hold the overhead required by the DynamicData internal representation (about 100 bytes) in addition to the minimum deserialized size of a sample. The minimum deserialized size of a sample assumes that all strings are allocated to their default values, sequences are left to length 0, and all optional members are unset. If set to any value other than 0: The underlying buffer will be allocated to the provided size plus the overhead required by the DynamicData internal representation (about 100 bytes). If the provided size plus the overhead is less than the size used when buffer_initial_size is left to 0, then the default value is used.</td>
</tr>
<tr>
<td>buffer_max_size</td>
<td>The maximum amount of memory that the underlying DynamicData buffer may use, in bytes. This property is used to configure the DynamicData objects that are created stand-alone as well as the DynamicData samples that are obtained from the sample pool that is created by each DynamicData DataReader. A DynamicData object will grow to this size from the initial size as needed. The buffer_max_size includes all overhead that is required for the internal DynamicData representation and therefore represents a hard upper limit on the size of the underlying DynamicData buffer. If set to -1 (default): The buffer will grow unbounded to the size required to fit all members. If set to any value other than -1: The buffer will not grow beyond this size. If setting a member’s values requires the buffer to grow beyond the maximum, the member will fail to be set. If the buffer is required to grow beyond this maximum during deserialization, the sample will fail to be deserialized. The buffer_max_size cannot be smaller than the buffer_initial_size.</td>
</tr>
</tbody>
</table>

Figure 22.4: Allocation of DDS Samples in DataReader Queue for DynamicData DataReaders below shows how DDS samples are allocated in the DataReader queue for DynamicData DataReaders.

Figure 22.4: Allocation of DDS Samples in DataReader Queue for DynamicData DataReaders
22.2.4 Memory Management for Fragmented DDS Samples

When a DataWriter writes DDS samples with a serialized size greater than the minimum of the largest transport message sizes across all transports installed with the DataWriter, the DDS samples are fragmented into multiple RTPS fragment messages.

The different fragments associated with a DDS sample are assembled in the DataReader side into a single buffer that will contain the DDS sample serialized data after the last fragment is received.

By default, the DataReader keeps a pool of pre-allocated serialization buffers that will be used to reconstruct the serialized data of a DDS sample from the different fragments. Each buffer hold one individual DDS sample and it has a size equal to the maximum serialized size of a DDS sample. The pool size can be configured using the QoS values initial_fragmented_samples and max_fragmented_samples in 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576.

The main disadvantage in pre-allocating the serialization buffers is an increase in memory usage, especially when the maximum serialized of a DDS sample is quite large. Connext DDS offers a setting that allows memory for a DDS sample to be allocated from the heap the first time a fragment is received. The amount of memory allocated equals the amount of memory needed to store all fragments in the DDS sample.

22.2.5 Reader-Side Memory Management when Working with Large Data

This section describes how to configure the DataReader side of the videoconferencing application introduced in 22.1.4 Writer-Side Memory Management when Working with Large Data on page 931 to optimize memory usage.

The following XML file can be used to optimize the memory usage in the previous example:

```xml
<?xml version="1.0"?>
<!-- XML QoS Profile for large data -->
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <!-- QoS Library containing the QoS profile used for large data -->
  <qos_library name="ReliableLargeDataLibrary">
    <!-- QoS profile used to optimize the memory usage in a DataWriter sending large data images -->
    <qos_profile name="ReliableLargeDataProfile" is_default_qos="true">
      <!-- QoS used to configure the DataWriter -->
      <datareader_qos>
        <history>
          <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <resource_limits>
          <max_samples>32</max_samples>
          <!-- No need to pre-allocate 32 frames unless needed -->
          <initial_samples>1</initial_samples>
        </resource_limits>
      </datareader_qos>
    </qos_profile>
  </qos_library>
</dds>
```

<!-- Since the video frame samples have a
large maximum serialized size we can configure the fragmented samples pool to use dynamic memory allocation. As an alternative, reduce max_fragmented_samples. However, that may cause fragment retransmission.

```xml
<reader_resourceLimits>
  <property>
    <value>
      <!-- Java will use a buffer of 33KB to deserialize all frames with a serialized size smaller or equal than 33KB. When an I-frame is received, Java will re-allocate the deserialization buffer to match the serialized size of the new frame. -->
      <element>
        <name>dds.data_reader.history.memory_manager.java_stream.min_size</name>
        <value>33792</value>
      </element>
      <element>
        <name>dds.data_reader.history.memory_manager.java_stream.trim_to_size</name>
        <value>1</value>
      </element>
    </value>
  </property>
</reader_resourceLimits>
```

To avoid preallocation of the samples in the DataReader's queue to their maximum size for Type-Plugin generated code in C, C++, Java, and .NET, replace the bounded sequence in VideoFrame with an unbounded sequence and generate code using the `-unboundedSupport` command-line option:

```c
struct VideoFrame {
  boolean keyFrame;
  sequence<octet> data;
};
```

See 22.2.1 Memory Management for DataReaders Using Generated Type-Plugins on page 934 for more details.

To avoid preallocation of the samples in the DataReader's queue to their maximum size for DynamicData, set the min_size_serialized property to avoid the allocation of 1MB buffers for the DataReader queue samples (See 22.2.3 Memory Management for DynamicData DataReaders on page 936.
22.3 Instance Memory Management for DataWriters

When an instance is registered with a `DataWriter`, the `DataWriter` serializes the key value and stores it with the instance.

Each instance maintained by the `DataWriter` has an associated buffer in which the `DataWriter` serializes the key. This buffer is either:

- Obtained from a pre-allocated pool (if the key’s serialized size is \( \leq \text{dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size} \)).
- Dynamically allocated from the heap (if the key’s serialized size is \( \geq \text{dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size} \)).

See Table 22.5 Instance Memory Management Properties for DataWriters.

### Table 22.5 Instance Memory Management Properties for DataWriters

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
</table>
| `dds.data_writer.history.memory_manager.fast_pool.pool_buffer_max_size` | Controls the memory allocation for the serialized key buffer that is stored with every instance. Default: -1 (UNLIMITED). All DDS sample buffers are obtained from the pre-allocated pool. The buffer size is the maximum serialized size of the DDS samples, as returned by the type plugin `get_serialized_sample_max_size()` operation. Notes:  
  - If you use unbounded sequences or strings as part of your key, then you should set `pool_buffer_max_size` to a finite value. See 3.1.1 Sequences on page 30.  
  - The `pool_buffer_max_size` also controls DDS sample memory management. See 22.1 DDS Sample Memory Management for DataWriters on page 926. |

22.4 Instance Memory Management for DataReaders

There are a number of features that require `DataReaders` to allocate a buffer to store an intermediate, serialized representation of a sample. The size of these buffers is controlled with the property `dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size`.

When a buffer is needed:

- If the required size is \( \leq \text{dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size} \), a preallocated buffer will be used. This buffer will not be freed until the `DataReader` is deleted and is therefore reused whenever a buffer is needed for a sample that matches this condition.
- If the required size is \( \geq \text{dds.data_reader.history.memory_manager.fast_pool.pool_buffer_max_size} \), the buffer will be dynamically allocated from the heap and then freed once it is no longer needed.
This property must be set if you are using any of the following features:

- Keyed data types
- Query conditions (only required if also using DynamicData, or the Java, .NET, or Modern C++ language APIs)
- Data encryption using *RTI Security Plugins*
- User-data payload compression (see 7.5.3.2 Data Compression on page 383)

In the case of keyed data types, a buffer is used per-instance to store the serialized representation of the key value for that instance.

In the case of query conditions, when a query condition is created, all samples that are in the *DataReader*'s queue need to be temporarily re-serialized in order to be evaluated against the query condition. A buffer that is allocated based on the value of this property is used for that serialization.

In the case of data encryption and compression, a buffer is needed when a sample is first received in order to decode or uncompress the sample into. If data encryption and compression are both being used, then two buffers will be allocated (because the sample must first be decoded into one buffer and then uncompressed into another buffer).

**Setting this property to a finite value**

The key buffers, used to store the serialized key per-instance, come from a pre-allocated pool with *ResourceLimits::initial_instances* initial buffers. Setting this property to a finite value will cause the buffers in the key buffer pool to be allocated to that size. If a larger buffer is needed when a new instance is received by the *DataReader*, a buffer of the correct size will be allocated at that time. If you use unbounded sequences or strings as part of your key, then you should set this property to a finite value. See 3.1.1 Sequences on page 30.

The buffers that are needed for samples are shared and are allocated once they are needed for the first time. When a buffer is needed, the required size will be checked against the value of this property. If the required size is less than or equal to this property's value and a buffer has not been allocated before, a buffer will be allocated with the property's size. This buffer will not be deallocated until the *DataReader* is deleted and will be reused every time a buffer with this size or smaller is needed. If a buffer is needed that is larger than the value configured by this property, it will be dynamically allocated to the correct size and then freed as soon as it is not needed anymore (once the sample has been deserialized). There will be at most two buffers allocated to the configured size. This will only happen if both data encryption and user-data payload compression are being used at the same time.

**Setting this property to unlimited (DEFAULT)**

If this property is set to -1 (UNLIMITED), the size of the key buffers is the maximum serialized size of the key as returned by the type plugin *get_serialized_key_max_size()* operation. These buffers still come from a pre-allocated pool. The size of the buffers used for samples will be allocated, when needed, to the
maximum serialized size of a sample as returned by the type plugin `get_serialized_sample_max_size()` operation. They are not deallocated until the `DataReader` is deleted so that they can be reused whenever needed.
Chapter 23 Sending Large Data

This section describes the capabilities offered by Connext DDS to allow sending and receiving large data samples. In this section, “large data” refers to samples with a large serialized size, usually on the order of MBs, such as video frame samples.

The definition of “large data” in this chapter contrasts with other definitions of large data in this manual:

- In 7.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) on page 345, “large data” is defined as data that cannot be sent as a single packet by a transport. The concept of large data in this section is decoupled from the maximum message size of the underlying transport, although these two things are related: samples with a size in the order of MBs will usually be greater than the underlying transport’s maximum message size.

- In Chapter 22 DDS Sample and Instance Memory Management on page 926, “large data” refers to types whose samples have a large maximum serialized size independently of the actual serialized size of the samples sent on the wire. This contrasts with the definition of “large data” in this section, which refers to samples with a large serialized size.

Connext DDS offers the following solutions to optimize the sending and receiving of large data:

- Reducing latency using either or both of the following to reduce the number of copies produced by the middleware; see 23.1 Reducing Latency on the next page:
  - RTI FlatData™ language binding; see 23.1.4 FlatData Language Binding on page 949
  - Zero Copy transfer over shared memory; see 23.1.5 Zero Copy Transfer Over Shared Memory on page 962

- Reducing bandwidth usage by compressing samples with a set of standard compression algorithms; see 7.5.3 DATA_REPRESENTATION QosPolicy on page 380
23.1 Reducing Latency

One of the main considerations when sending large samples is latency. When using Connext DDS, and in general any connectivity framework, sample latency has three components: middleware, copy, and transport (see Figure 23.1: Basic Components of Latency below).

Figure 23.1: Basic Components of Latency

- **Middleware**: Operational latency introduced by the middleware to provide functionality that otherwise would have to be provided by the application, such as durability and instance management.

- **Copy**: Latency introduced by copying the sample content. This includes the serialization (marshaling) and deserialization (unmarshaling) of copies.

- **Transport**: Latency introduced by the underlying transport and networking infrastructure.

When Connext DDS is used to send small data samples, such as temperature readings, the weight of the copy component in the total sample latency is small. But when samples are large, the weight of the copy component becomes considerable. (See Figure 23.2: Copy Components Compared on the next page.)
Therefore, reducing the number of copies made by the middleware or network infrastructure when publishing and receiving large samples becomes critical. Two features allow reducing the number of sample copies and consequently the transmission latency: Zero Copy transfer over shared memory and FlatData language binding. These two features can be used individually or in combination.

**Important:** “Large samples” in this context refers to samples with a large serialized size, usually on the order of MBs, such as video frame samples. If you implement FlatData language binding or Zero Copy transfer over shared memory with data smaller than this, you may not see significant difference in latency or even pay a penalty in latency.

### 23.1.1 Use Cases

Zero Copy transfer over shared memory and FlatData language binding are recommended when your strict latency requirements cannot be met by regular C/C++ language binding (which defines the in-memory representation of a type), and the UDP and shared memory transports. For example, video applications such as video conferencing, video surveillance, or computer vision usually have strict latency requirements, especially if the video signal is used to do control. Consider, for instance, a latency requirement of less than 100 milliseconds. This latency must account for different components such as:

- Video compression
- Video decoding
- Transmission
23.1.2 Copies in the Middleware Memory Space

- Image scaling
- Application processing logic

![Diagram of video processing stages with 100 msec delay]

To keep latency to a minimum for large data samples, reduce the number of copies made by the middleware or network infrastructure by using FlatData language binding, Zero Copy transfer over shared memory, or both.

23.1.2 Copies in the Middleware Memory Space

Figure 23.3: Number of Copies Out-of-the-Box on the next page shows how many times Connext DDS may copy a large sample sent over UDP or shared memory. The diagram assumes that the samples have to be fragmented by the middleware (via DDS fragmentation) because their serialized size is greater than the underlying transport MTU (maximum transmission unit), which can be configured by setting `message_size_max` in the transport properties (see Chapter 16 Transport Plugins on page 796). Note that these are copies in the middleware memory space—the operating system network stack may make additional copies.
For both UDP and shared memory (SHMEM), the copies are as follows, out of the box:

1. Copy 1 is the serialization copy. Connext DDS calls **TypePlugin::serialize** to convert the in-memory representation of a sample, such as a C++ object, into a data representation, called a wire representation, with a format suitable for storage or transmission.

After a sample is serialized, it is sent to the subscribing application using one or more of the available transports. When the underlying transport maximum message size is smaller than the serialized size of the sample, the sample must be fragmented. The fragmentation process does not require any extra copy. Fragments refer directly to offsets in the serialization buffer.

**Note:** The transport maximum message size can be configured using the property **dds.transport.UDPv4.builtin.parent.message_size_max** for UDPv4 and **dds.transport.shmem.builtin.parent.message_size_max** for SHMEM. There are equivalent properties for other transports, such as TCPv4 and UDPv6.
2. Copy 2: For SHMEM, the sample fragments that live in the local memory space of the publishing process have to be copied into the shared memory segment from which the subscribing application will read them. For UDP, the call to the socket receive operation copies the fragments.

3. Copy 3: After they are received, the sample fragments are reassembled into a single buffer.

4. Copy 4 is the deserialization copy. Connext DDS calls `TypePlugin::deserialize` to convert the wire memory representation of a sample into its in-memory representation, such as a C++ object.

Figure 23.4: Number of Copies Using FlatData Language Binding

FlatData is a language binding in which the in-memory representation of a sample matches the wire representation. Therefore, the cost of serialization/deserialization is zero. You can directly access the serialized data without deserializing it first. When using FlatData language binding, Copy 1 and Copy 4 in Figure 23.3: Number of Copies Out-of-the-Box on the previous page are removed for both UDP and SHMEM communications. See Figure 23.4: Number of Copies Using FlatData Language Binding above.
23.1.3 Choosing between FlatData Language Binding and Zero Copy Transfer over Shared Memory

Whether to use Zero Copy transfer over shared memory or FlatData language binding, or both, depends on whether the DataReaders run on the same host as the DataWriters, on different hosts, or a combination of both. It also depends on the definition of the type. Zero Copy transfer over shared memory requires the FlatData language binding when the type is variable-size. The following table summarizes how to choose between these features:

<table>
<thead>
<tr>
<th>Fixed-size types</th>
<th>Readers and writers run on same host</th>
<th>Readers and writers run on different hosts</th>
<th>Some readers/writers run on same host, some on different hosts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Zero Copy</td>
<td>Use FlatData</td>
<td>Use both Zero Copy and FlatData</td>
<td></td>
</tr>
<tr>
<td>Variable-size types</td>
<td>Use both Zero Copy and FlatData</td>
<td>Use FlatData</td>
<td>Use both Zero Copy and FlatData</td>
</tr>
</tbody>
</table>

In summary, for DataReaders running on the same host as the DataWriter, the DataWriter can take advantage of Zero Copy transfer over shared memory. For DataReaders running on a different host, the DataWriter won’t use Zero Copy transfer over shared memory, but can benefit from FlatData language binding. Therefore, when you have writers and readers running on the same and different hosts, it is recommended to use both Zero Copy transfer over shared memory and FlatData language binding, and let the DataWriter use the correct option for each DataReader.

For more information, see 23.1.4 FlatData Language Binding below and 23.1.5 Zero Copy Transfer Over Shared Memory on page 962.

23.1.4 FlatData Language Binding

FlatData language binding offers the following benefits:

- Reduced number of copies: from four to two for both SHMEM and UDP transports (see Figure 23.4: Number of Copies Using FlatData Language Binding on the previous page), because there is no need to serialize and deserialize a sample.
- Reduced memory consumption and CPU load, due to reduced data copying.
- Improved latency for large data samples.

23.1.4.1 FlatData Representation

When you create a FlatData sample (see 23.1.4.2.2 Programming with FlatData Language Binding on page 952), the in-memory representation for the sample buffer is XCDR encoding version 2 (XCDR2), using the endianness of the host where the sample is created to populate the buffer. The use of the host
platform endianness allows fast access to the sample content, because the setters and getters do not have to change the endianness.

If you use a DataReader to read a FlatData sample that was received from a DataWriter running on a platform with a different endianness, however, direct access to the sample content is not possible, making the subscribing application less performant.

**Note:** Because the in-memory representation of a FlatData sample is XCDR2 and older versions of Connext DDS use encoding version 1 (XCDR), applications using the FlatData language binding will not communicate with older versions of Connext DDS. See Choosing the Right Data Representation, in the Data Representation chapter of the RTI Connext DDS Core Libraries Extensible Types Guide.

### 23.1.4.2 Using FlatData Language Binding

The following sections contain more information about using FlatData language binding:

- 23.1.4.2.1 Selecting FlatData Language Binding below
- 23.1.4.2.2 Programming with FlatData Language Binding on page 952
- 23.1.4.3 Languages Supported by FlatData Language Binding on page 961
- 23.1.4.3.1 Interactions with RTI Security Plugins and Compression on page 961
- 23.1.4.3.2 Notes on Batching on page 962

For examples of FlatData language binding and Zero Copy transfer over shared memory, including example code, see https://community.rti.com/kb/flatdata-and-zero-copy-examples.

#### 23.1.4.2.1 Selecting FlatData Language Binding

To select FlatData as the language binding of a type, annotate it with @language_binding(FLAT_DATA). (See 3.3.9.9 The @language_binding Annotation on page 120.)

For example, consider a surveillance application in which high-definition (HD) video signal is published and subscribed to. The application publishes a Topic of the type CameraImage. This is the IDL:

```c
enum Format {
    RGB,
    HSV,
    YUV
};

@final
@language_binding(FLAT_DATA)
struct Resolution {
    int32 height;
    int32 width;
};
```
The language binding annotation supports two values: FLAT_DATA and PLAIN (default). PLAIN refers to the regular in-memory representation, where an IDL struct maps to a C++ class or C struct.

There are some restrictions regarding the kinds of structures, value types, and unions to which the FlatData language binding can be applied.

For final types, the FlatData language binding can be applied only to fixed-size types. A fixed-size type is a type whose wire representation always has the same size. This includes primitive types, arrays of fixed-size types, and structs containing only members of fix-size types. Unions are not fixed-size types.\(^1\)

The FlatData language binding can be applied to any mutable type. This enables support for variable-size types containing bounded sequences, bounded strings, or optional members (unbounded sequences or strings are not supported with FlatData). It also allows using unions.

FlatData cannot be applied to extensible types.

Final types provide the best performance, while mutable types are the most flexible. Typically, the best compromise between flexibility and performance comes from a mutable type whose largest member is either a final type or a sequence of final elements. In the `CameraImage` example, the top-level type is mutable, which allows for type evolution, optional members, and variable-size members (such as the source string member). On the other hand, its member `pixels`, which contains the bulk of the data, is defined as a sequence of the final type `Pixel`, which allows for an efficient manipulation.

---

\(^1\) These restrictions on final types only apply to the FlatData language binding. Final types with the plain language binding can be variable-size.
23.1.4.2.2 Programming with FlatData Language Binding

When a type is marked with the FlatData language binding, the in-memory representation for samples of this type is equal to the wire representation (according to XCDR version 2\(^1\)). That is, the data sample is in its serialized format at all times. To facilitate accessing and setting the sample content, RTI Code Generator generates helper types that provide the operations to create and access these data samples. These helper types are Samples, Offsets, and Builders.

A FlatData Sample is a buffer holding the wire representation of the data. In the code generated for the previous IDL, a sample of the type CameraImage contains this buffer. This is the top-level object that can be written or read:

```
typedef rti::flat::Sample<CameraImageOffset> CameraImage;
```

(Note: These examples show code for the Modern C++ API. See 23.1.4.3 Languages Supported by FlatData Language Binding on page 961.)

To access this sample, applications use Offset types. An Offset represents the type of a member and its location in the buffer. An Offset can be described as an “iterator,” a light-weight object that points to the data, but doesn’t own it. Copying an Offset copies the “iterator,” not the data it points to.

```cpp
class NDDSUSERDLlExport CameraImageConstOffset : public rti::flat::MutableOffset {
    public:
        const rti::flat::StringOffset source() const;
        Format format() const;
        Resolution::ConstOffset resolution() const;
        rti::flat::SequenceOffset<Pixel::ConstOffset> pixels() const;
    };

class NDDSUSERDLlExport CameraImageOffset : public rti::flat::MutableOffset {
    public:
        typedef CameraImageConstOffset ConstOffset;

        // Const accessors
        const rti::flat::StringOffset source() const;
        Format format() const;
        Resolution::ConstOffset resolution() const;
        rti::flat::SequenceOffset<Pixel::ConstOffset> pixels() const;

        // Modifiers
        rti::flat::StringOffset source();
        bool format(Format value);
        Resolution::Offset resolution();
        rti::flat::SequenceOffset<Pixel::Offset> pixels();
    };
```

There are two kinds of Offset types:

---

\(^1\) See Data Representation, in the RTI Connext DDS Core Libraries Extensible Types Guide for more information on XCDR2.
• Generated, named Offsets, to access a user-defined struct or union type (CameraImageOffset, PixelOffset). They provide accessors to directly get or set primitive members, and one getter for each non-primitive member to retrieve its corresponding Offset.

Each named offset has a corresponding read-only version (CameraImageConstOffset). This is analogous to a read-only iterator (e.g., std::vector<T>::const_iterator and std::vector<T>::iterator).

• Offsets to arrays, sequences, strings, and other IDL types. They provide access to their elements. Primitive elements can be accessed directly; non-primitive elements are accessed through Offsets for their types.

For details on all the Offset types and their interface, see the API Reference HTML documentation, under RTI Connext DDS API Reference > Topic Module > FlatData Topic-Types.

The function CameraImage::root() provides the Offset to the top-level type (CameraImageOffset). If the sample is const (for example, in a LoanedSamples container), root() returns a read-only offset (CameraImageConstOffset).

![Diagram of CameraImage structure]

To create variable-size (mutable) data-samples, applications use Builders. A Builder type provides the interface to create a mutable sample member by member. Once all the desired members for a sample have been added, the Builder is “finished,” returning the built sample, which can be published.

```cpp
class NDDSUSERDllExport CameraImageBuilder : public rti::flat::AggregationBuilder {
public:
    typedef CameraImageOffset Offset;

    Offset finish();
    CameraImage * finish_sample();

    rti::flat::StringBuilder build_source();
    bool add_format(Format value);
    Resolution::Offset add_resolution();
    rti::flat::FinalSequenceBuilder<Pixel::Offset> build_pixels();
};
```

Builders provide three kinds of functions:

• **add_<member>** functions insert a member of a final type, returning an Offset to it.

• **build_<member>** functions provide another Builder to create a member of a mutable type.
• **finish** and **finish_sample** end the construction of a member or a sample, respectively.

Similarly to Offsets, Builders can correspond to user-defined struct and union types, or other IDL types such as sequences, arrays, and strings. For details on all the Builder types see the API Reference HTML documentation.

The following sections summarize how to use FlatData language binding:

• Creating a FlatData sample below
• Writing a FlatData sample on page 957
• Reading a FlatData sample on page 957
• Working with unmanaged FlatData samples on page 958
• Multi-threading notes on page 959
• Notes on extensible types on page 960

Creating a FlatData sample

The following sections assume you have created a *DataWriter* for the type **Pixel** or **CameraImage**, following the usual process.

To write FlatData, first create a FlatData sample. The way to create a sample varies depending on whether the type is final or mutable. In both cases, this section shows how to create *DataWriter*-managed samples. See also Working with unmanaged FlatData samples on page 958.

**Creating a FlatData sample for a final type**

In this section we will create a sample for the final type **Pixel**. To create a sample for the mutable type **CameraImage**, see Creating a FlatData sample for a mutable type after this.

Samples for final FlatData types are created directly with a single call to the *DataWriter* function **get_loan**. The *DataWriter* manages this sample and will return it to a pool at some point after the sample is written.

```c
Pixel *pixel_sample = writer.extensions().get_loan();
```

**pixel_sample** contains the buffer that can be written. To set its values, first locate the position of the top-level type:
The `root()` function returns `PixelOffset`, which points to the position where the data begins. To set the values, use the following setters:

```cpp
pixel.red(10);
pixel.green(20);
pixel.blue(30);
```

### Creating a FlatData sample for a mutable type

Samples for mutable types are created using Builders. To obtain a `CameraImageBuilder` to build a `CameraImage` sample, use the function `build_data`:

```cpp
CameraImageBuilder image_builder = rti::flat::build_data(writer);
```

This function loans the memory necessary to create a `CameraImage` sample from the `DataWriter` and provides a `CameraImageBuilder` to populate it.

Use the Builder functions to set the sample’s members in any order. For a FlatData type that is also mutable, such as this one, you must call methods ("add" or "build") to set all the members you want to be sent. If you don't set a member's value, that member won't be sent by Connext DDS. If a member is a key of the data-type, you must set its value; otherwise the `DataWriter`'s write operation will fail. Effectively, all non-key members of a FlatData, mutable data type are treated as if they were `@optional`. (See Optional Members, in the Type System Enhancements chapter of the RTI Connext DDS Core Libraries Extensible Types Guide.)

These Builder functions work on a pre-allocated buffer; they do not allocate any additional memory.

First, we add the member `format`. As a primitive member, the function `add_format` directly adds the member and sets its value:

```cpp
image_builder.add_format(Format::RGB);
```

Next, we add the member `resolution`. Its type being final, the function `add_resolution` adds the member and provides the Offset that allows setting its values:

```cpp
ResolutionOffset resolution = image_builder.add_resolution();
resolution.height(100);
resolution.width(200);
```

To build the string member `source`, the function `build_source` returns a `StringBuilder`. We use this builder (in this case it’s as simple as calling `set_string`), and then call `finish`. The function `finish` (not to be confused with `finish_sample`) completes the construction of the member and renders `source_builder` invalid.

```cpp
auto source_builder = image_builder.build_source();
source_builder.set_string("CAM-1");
source_builder.finish();
```

Since this builder is so simple, it is possible to simplify the above code:
image_builder.build_source().set_string("CAM-I");

(The Builder destructor takes care of calling \texttt{finish}.)

To create the \texttt{pixels} member, we build a sequence of \texttt{Pixels}:

\begin{verbatim}
auto pixels_builder = image_builder.build_pixels();
\end{verbatim}

There are two ways to populate this member.

\textit{Method 1:} add and initialize each element:

\begin{verbatim}
for (int i = 0; i < 20000; i++) {
    PixelOffset pixel = pixels_builder.add_next();
    pixel.red(i % 256);
    pixel.green((i + 1) % 256);
    pixel.blue((i + 2) % 256);
}
pixels_builder.finish();
\end{verbatim}

Builders for sequences with elements of a final type provide the function \texttt{add\_next} to add the elements. When the element type is mutable, the sequence (and array) Builder provides the function \texttt{build\_next}, which provides a Builder for each element. See more details in the API Reference HTML documentation.

\textit{Method 2:} cast the elements in the sequence to the equivalent C++ plain type. This method only works for types that meet the conditions required by \texttt{rti::flat::plain\_cast}, as described in the API Reference HTML documentation. Basically, the in-memory representation must match the XCDR2 serialized representation. \texttt{Pixel} meets these conditions.

Method 2 is more efficient. First, we use the Builder function \texttt{add\_n} to add 20000 elements at once, leaving them uninitialized. Then, after finishing the Builder, we obtain the Offset to the member, cast it, and manipulate the data as a plain C++ type:

\begin{verbatim}
pixels_builder.add_n(20000);
auto pixels_offset = pixels_builder.finish();

auto plain_pixels = rti::flat::plain_cast(pixels_offset);
for (int i = 0; i < 20000; i++) {
    plain_pixels[i].red(i % 256);
    plain_pixels[i].green((i + 1) % 256);
    plain_pixels[i].blue((i + 2) % 256);
}
\end{verbatim}

The function \texttt{rti::flat::plain\_cast} casts the position in memory that \texttt{pixels\_offset} points to into a C-style array of \texttt{PixelPlainHelper}, a type with the same IDL definition as \texttt{Pixel}, but with \texttt{@language\_binding (PLAIN)}. \texttt{plain\_cast} can receive an offset to a final struct, or an offset to an array or sequence of final structs or primitive types. See the API Reference HTML documentation for more information.

Finally, call \texttt{finish\_sample} to obtain the complete sample. After this, the Builder instance is invalid and cannot be further used.
CameraImage *image_sample = image_builder.finish_sample();

Once the sample has been created, it is still possible to modify its values, as long as these modifications don’t change the size. For example, it is possible to change the value of an existing pixel, but it’s not possible to add a new one:

```cpp
auto pixels_offset = image_sample->root().pixels();
pixels_offset.get_element(100).blue(0);
```

The next section shows how to write the sample.

### Writing a FlatData sample

When you write a sample using a regular *DataWriter* (for a type with a plain language binding), the *DataWriter* copies the sample in its internal queue, so when *write()* ends, the application still owns the sample. A *DataWriter* for a FlatData type, however, doesn’t copy the sample; it keeps a reference. You yield ownership of the data sample from the moment you call *write()*:

```cpp
writer.write(*image_sample);
```

The *DataWriter* will decide when to return samples created with *get_loan* or *build_data* to a pool, where the sample will be reused.

To write a new sample, don’t use *image_sample* again, but obtain a new one with *get_loan* or build a new one with *build_data*.

If the sample cannot be written, to return it to the *DataWriter* pool call:

```cpp
writer.extensions().discard_loan(*image_sample);
```

Or, if the sample has not been completely built yet, discard the Builder:

```cpp
rti::flat::discard_builder(writer, image_builder);
```

### Reading a FlatData sample

The method for reading data for a FlatData type is the same regardless of whether the type is final or mutable.

Create a *DataReader* as you normally would; see 8.3.1 Creating DataReaders on page 510.

Read the data samples:

```cpp
dds::sub::LoanedSamples<CameraImage> samples = camera_reader.take();
```

Let’s work with the first sample (assuming *samples.length() > 0* and *samples[0].info().valid()*):

```cpp
const CameraImage& image_sample = samples[0].data();
```

Using the **root** Offset and the Offset to the members, the following code prints the sample values. Note that in this example, *image_sample* is **const**, so *camera_image* is a *CameraImageConstOffset*, which only allows reading the buffer, not modifying it.
auto camera_image = image_sample->root();

std::cout << "Source: " << camera_image.source().get_string() << std::endl;
std::cout << "Timestamp: " << camera_image.timestamp() << std::endl;
std::cout << "Format: " << camera_image.format() << std::endl;

auto resolution = camera_image.resolution();
std::cout << "Resolution (height: " << resolution.height() << "), width: " << resolution.width() << ")n";

Some members of image_sample may not exist. For example, if the field resolution wasn't received, then resolution.is_null() is true; if timestamp is not received, camera_image.timestamp() returns 0.

To access the sequence of pixels, the same two methods that allowed building it (element by element or plain cast) are available:

Method 1 (access each element offset):

```cpp
for (auto pixel : camera_image.pixels()) {
  std::cout << "Pixel (" << pixel.red() << ", " << pixel.green()
             << ", " << pixel.blue() << ")\n";
}
```

**Method 2 (plain_cast):**

```cpp
auto pixel_count = camera_image.pixels().element_count();
auto plain_pixels = rti::flat::plain_cast(camera_image.pixels());
for (int i = 0; i < pixel_count; i++) {
  const auto& pixel = plain_pixels[i];
  std::cout << "Pixel (" << pixel.red() << ", " << pixel.green()
             << ", " << pixel.blue() << ")\n";
}
```

Method 2 is more efficient, provided that the type meets the requirements of plain_cast. Also, the endianness of the publishing application must be the same as the local endianness.

Note that you can directly print the sample:

```cpp
std::cout << *image_sample << std::endl;
```

### Working with unmanaged FlatData samples

The previous sections describe how to create and write DataWriter-managed samples (via get_loan or build_data). While this is the recommended and easiest way, sometimes applications may need to use unmanaged samples. For example, they may need to reuse the same sample after it is written or to obtain the memory from some other source.

Note that a given DataWriter cannot write both unmanaged and managed samples. The functions get_loan or build_data will fail if an unmanaged sample has been written. Conversely, the DataWriter will fail to write an unmanaged sample if get_loan or build_data have been called.

To create a CameraImage using memory from an arbitrary buffer, my_buffer, with a capacity of my_buffer_size bytes, use the following constructor:
unsigned char *my_buffer = ...;
unsigned int my_buffer_size = ...;
CameraImageBuilder image_builder(my_buffer, my_buffer_size);
// use image_builder...
CameraImage *image_sample = image_builder.finish_sample();

**image_builder** will fail if it runs out of space. The maximum size of a **CameraImage** can be obtained from its dynamic type:

```cpp
unsigned int max_size =
rti::topic::dynamic_type<CameraImage>::get().cdr_serialized_sample_max_size();
```

After writing **image_sample**, the **DataWriter** takes ownership of it. In order to reuse the sample, the application needs to monitor the **on_sample_removed** callback in the **DataWriter** listener, and correlate the cookie it receives with the sample. The following is a simple **DataWriterListener** implementation that does that:

```cpp
class FlatDataWriterListener
: public dds::pub::NoOpDataWriterListener<CameraImage> {
public:
    void on_sample_removed(
        dds::pub::DataWriter<CameraImage>& writer,
        const rti::core::Cookie& cookie) override
    {
        // The cookie identifies the sample being removed
        last_removed_sample = cookie.to_pointer<CameraImage>();
    }
}
```

CameraImage *last_removed_sample = NULL;
```

The application will need to wait until **last_removed_sample** is equal to **image_sample**. This indicates that the **DataWriter** no longer needs to hold ownership of **image_sample**.

Another way to create an unmanaged sample is **CameraImage::create_data()** or **Pixel::create_data()** (the result of **CameraImage::create_data()** must be passed to the **CameraImageBuilder** constructor mentioned before). Samples can be copied with the **clone()** function. These samples need to be released with the respective **delete_data()** functions. See the API Reference HTML documentation for more information.

**Multi-threading notes**

- It’s not safe to use the same Offset object in parallel, even for reading. For efficiency, each offset object contains an internal state that may change when accessing a member.

```cpp
void my_thread1(CameraImageOffset& camera_image) {
    auto format = camera_image.format();
}
```

```cpp
void my_thread2(CameraImageOffset& camera_image) {
    // ...
}
```
It is safe to use different Offset objects to read the same member in a sample.

// Safe
auto camera_image1 = camera_image_sample.root();
auto camera_image2 = camera_image_sample.root();
std::async(my_thread1, camera_image1);
std::async(my_thread2, camera_image2);

It is not safe to build a sample using a Builder in parallel.

Notes on extensible types

There are a few differences in how a plain and a FlatData DataReader behave when they receive samples of types that are different but compatible.

Before a DataReader and DataWriter can communicate, their types are inspected to determine if they are compatible. The same is true when using FlatData; however, even after two types have been deemed compatible, there may be specific data samples that are not.

DataReaders for plain types verify sample compatibility during data deserialization, but DataReaders for FlatData types don’t deserialize the data, passing FlatData samples directly to the application. For that reason, there may be situations where a plain DataReader would lose or reject a data-sample, while a DataReader for a FlatData type with the same definition will pass the same sample to the application. Therefore, if you are using FlatData you may need to explicitly check if all the received samples are consistent with your application logic. For more information on the rules that determine the assignability of a sample, see the RTI Connext DDS Core Libraries Extensible Types Guide (see the section “Verifying Sample Consistency: Sample Assignability”) or the OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3.

For example, a FlatData DataReader won’t drop a sample when a sequence (or a string) member exceeds the bounds in the reader’s type definition, and the application will be able to read this sequence (or string). This can only happen if ignore_sequence_bounds (or ignore_string_bounds) in TypeConsistencyEnforcement is set to true; otherwise the DataWriter’s type won't match the DataReader’s. The @min and @max annotations are another example. FlatData DataReaders will not enforce the @min/@max range set for a member, and applications will be able to access such samples.

Another difference in behavior involves the reception of samples that don’t include some data members. When a regular DataReader for a mutable (plain) type receives a data sample that doesn’t include one of its non-optional members, it automatically assigns a default value during the deserialization. A
FlatData **DataReader** for a mutable (FlatData) type will not do that. Instead, if the application tries to access that member, the corresponding member getter will return a null Offset. Only if the member is primitive will it return a default value. This means that, for a FlatData **DataReader** in this case, all non-primitive members will be treated as if they were optional.

### 23.1.4.3 Languages Supported by FlatData Language Binding

The FlatData language binding is supported in the Modern and Traditional C++ APIs:

- `rtiddsgen -language C++11` generates code for the Modern C++ API.
- `rtiddsgen -language C++` generates code for the Traditional C++ API.

The FlatData language binding is basically the same in both APIs, as described in the previous sections, with a few differences:

- Modern C++ may throw exceptions in Sample, Offset, and Builder operations, such as `dds::core::PreconditionNotMetError`; Traditional C++ doesn’t throw exceptions and in these cases it would return invalid objects. See the API Reference HTML documentation for each language for details.
- Modern C++ maps integer types to int32_t, uint16_t, etc; Traditional C++ uses DDS_Long, DDSUnsignedShort, etc. This is consistent with these languages’ respective plain language bindings.
- Modern C++ provides an overloaded `operator<<` to print a sample; Traditional C++ uses `FooTypeSupport::print_data`. Both provide a function to transform to a string with format options. This behavior is also consistent with the plain binding.

#### 23.1.4.3.1 Interactions with RTI Security Plugins and Compression

When the FlatData language binding is used in combination with either payload encryption or compression (see [7.5.3 DATA REPRESENTATION QosPolicy on page 380](#)), there is no reduction in the number of copies used to send or receive the samples. There are unavoidable copies that must be made during the encryption/decryption and/or compression/decompression processes, resulting in the same number of copies that would be made if you were using regular data in combination with these features. This is a known issue that will be addressed in future releases (see "Known Issues" in the [RTI Connext DDS Core Libraries Release Notes](#)).

Therefore, using the FlatData language binding in combination with payload encryption and/or compression is not generally useful. However, when using Zero Copy transfer over shared memory, you will also need to use FlatData for variable-sized data types. (See [23.1.3 Choosing between FlatData Language Binding and Zero Copy Transfer over Shared Memory on page 949](#).) In this case, you may want to configure the **DataWriter** to use encryption/compression when sending to **DataReaders** running on different hosts, even though you are not saving copies. This configuration allows using Zero Copy for **DataReaders** running on the same host as the **DataWriter**, while encrypting/compressing data that is sent to **DataReaders** on different hosts.
23.1.5 Zero Copy Transfer Over Shared Memory

23.1.4.3.2 Notes on Batching

A FlatData DataWriter (a DataWriter that sends FlatData samples) cannot batch samples. That is, Connext DDS will not let you set up a FlatData DataWriter to use batching. Both FlatData and regular DataReaders, however, can receive batched samples from a regular DataWriter as well as all samples from a FlatData DataWriter.

23.1.5 Zero Copy Transfer Over Shared Memory

For communication within the same node using the built-in shared memory transport, by default Connext DDS copies a sample four times (see Figure 23.3: Number of Copies Out-of-the-Box on page 947). FlatData language binding reduces the number of copies to two (see Figure 23.4: Number of Copies Using FlatData Language Binding on page 948): the copy of the sample into the shared memory segment in the publishing application and the copy to reassemble the sample in the subscribing application. Two copies, however, may still be too many depending on the sample size and system requirements.

Zero Copy transfer over shared memory, provided as a separate library called nddsmept, allows reducing the number of copies to zero for communications within the same host. The nddsmept library can be linked with Connext DDS C or C++ libraries. This feature accomplishes zero copies by using the shared memory (SHMEM) built-in transport to send 16-byte references to samples within a SHMEM segment owned by the DataWriter, instead of using the SHMEM built-in transport to send the serialized sample content by making a copy. See Figure 23.5: Zero Copy Transfer Over Shared Memory below.

With Zero Copy transfer over shared memory, there is no need for the DataWriter to serialize a sample, and there is no need for the DataReader to deserialize an incoming sample since the sample is accessed directly on the SHMEM segment created by the DataWriter.

Figure 23.5: Zero Copy Transfer Over Shared Memory
This feature offers the following benefits:

- Number of copies is reduced from four to zero (see SHMEM in Figure 23.3: Number of Copies Out-of-the-Box on page 947). Instead of transferring the entire sample by making multiple copies, only the location in shared memory is distributed to DataReaders (see Figure 23.5: Zero Copy Transfer Over Shared Memory on the previous page).

- Because of this reduced data copying, memory consumption and CPU load are also reduced.

- Latency is independent of the size of the sample.

- Fragmentation is not required when using Zero Copy transfer over shared memory because the DataWriter exchanges SHMEM references (only 16-bytes) with DataReaders and not the full sample.

- Data can still be sent off-board, simplifying application deployment and configuration. When the data is sent off-board, the middleware is still making the same copies described in Figure 23.3: Number of Copies Out-of-the-Box on page 947. To reduce the number of copies for sending off-board, use FlatData language binding in conjunction with Zero Copy transfer over shared memory.

Note: A Zero Copy DataWriter is defined as any DataWriter with the ability to send a sample reference. You can have a DataWriter that does both: sends sample references to Zero Copy DataReaders, and sends serialized samples to non-Zero Copy DataReaders. In this case, the DataWriter is still considered a Zero Copy DataWriter in this documentation.

**23.1.5.1 Using Zero Copy Transfer Over Shared Memory**

To use Zero Copy transfer over shared memory, perform the following basic steps:

- Identify types that require Zero Copy transfer over shared memory and annotate them with @transfer_mode(SHMEM_REF) in the IDL files. (See: 3.3.9.8 The @transfer_mode annotation on page 119.)

  Note: Zero Copy transfer over shared memory requires the FlatData language binding when the type is variable-size.

- Use the DataWriter’s get_loan() API to get a loaned sample for writing with Zero Copy. (You would use this API to create the sample rather than creating the sample using the TypeSupport. See the example in the following sections and the API Reference HTML documentation for more information on get_loan().)

- Link the publisher and subscriber application with the additional Zero Copy library, nddsmetp. (RTI Code Generator (rtiddsgen) generates examples that link nddsmetp for you automatically. If you are using a custom build system, make sure you link with nddsmetp.)
RTI Code Generator generates additional TypePlugin code when a type is annotated with @transfer_mode(SHMEM_REF) in the IDL files. This code allows a DataWriter and a DataReader to communicate using a reference to the sample in shared memory (see Figure 23.5: Zero Copy Transfer Over Shared Memory on page 962). In addition to sending a sample reference, the DataWriter can also send the serialized sample to a DataReader that doesn’t support Zero Copy transfer over shared memory.

The following sections contain more information about using Zero Copy transfer over shared memory:

- 23.1.5.1.1 Sending data with Zero Copy transfer over shared memory below
- 23.1.5.1.2 Receiving data with Zero Copy transfer over shared memory on page 966
- 23.1.5.1.3 Checking data consistency with Zero Copy transfer over shared memory on page 966
- 23.1.5.1.4 Languages Supported by Zero Copy Transfer Over Shared Memory on page 967
- 23.1.5.1.5 Interactions with RTI Security Plugins and Compression on page 968
- 23.1.5.1.6 Notes on Batching on page 968

For examples of FlatData language binding and Zero Copy transfer over shared memory, including example code, see https://community.rti.com/kb/flatdata-and-zerocopy-examples.

### 23.1.5.1.1 Sending data with Zero Copy transfer over shared memory

The following example shows how to use Zero Copy transfer mode for a surveillance application in which high-definition (HD) video signal is published and subscribed to. The application publishes a Topic of the type CameraImage. This is the IDL:

```idl
enum Format {
    RGB,
    HSV,
    YUV
};

struct Resolution {
    int32 height;
    int32 width;
};

const long IMAGE_SIZE = 8294400 * 3;

@transfer_mode(SHMEM_REF)
struct CameraImage {
    int64 timestamp;
    Format format;
    Resolution resolution;
    octet data[IMAGE_SIZE];
};
```
The CameraImage type is annotated with @transfer_mode(SHMEM_REF) to allow Zero Copy communication. Note that it is sufficient to annotate only top-level types with this annotation.

Any final or appendable type annotated with @transfer_mode(SHMEM_REF) should be a fixed-size type. This means the type can include primitive members, arrays of fixed-size types, and structs containing only members of fixed-size types. To use a variable-sized type, the type should be annotated with @language_binding(FLAT_DATA) and @mutable in combination with @transfer_mode(SHMEM_REF).

With Zero Copy transfer mode, an application writes samples coming from a shared memory sample pool created by a Zero Copy DataWriter. Therefore, create a DataWriter before creating a sample. The steps for creating a Zero Copy DataWriter are the same as for a regular DataWriter.

```cpp
    const int MY_DOMAIN_ID = 0;
    dds::domain::DomainParticipant participant(MY_DOMAIN_ID);
    dds::topic::Topic<CameraImage> camera_topic(participant, "Camera");
    dds::pub::DataWriter<CameraImage> camera_writer(                  
        rti::pub::implicit_publisher(participant),                   
        camera_topic);
```

To get a sample from shared memory, use the DataWriter’s get_loan() API:

```cpp
    CameraImage *camera_image = camera_writer->get_loan();
```

The sample returned by get_loan() is uninitialized by default (the members are not set to default values). If you would like to allow the DataWriter to return an initialized sample from get_loan(), set initialize_writer_loaned_sample to true in the 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 401.

Populate the fields of the sample as you would a regular sample:

```cpp
    camera_image->timestamp(12345678);
    camera_image->format(Format::HSV);
    camera_image->resolution().height(1024);
    camera_image->resolution().width(2048);
    // populate the image data
```

The example above, showing the population of the fields, assumes regular PLAIN language binding. Zero Copy transfer over shared memory also works with types using FLAT_DATA language binding. In this case, you must use the FlatData API described in 23.1.4 FlatData Language Binding on page 949 to populate the sample.

The number of samples in the shared memory sample pool created by the DataWriter can be configured using the writer_loaned_sample_allocation settings in the 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 401.

Initially all the samples are in a free state. When you call the DataWriter’s get_loan(), the DataWriter provides a sample from this pool, and its state changes to allocated. The samples are provided using an LRU (Least Recently Used) policy.

Write the sample with the regular write operation:
When a sample is written, its state transitions from allocated to enqueued, and the `DataWriter` takes responsibility for returning the sample back to the shared memory pool. The sample remains in the enqueued state until it is removed from the `DataWriter` queue. When this happens, the sample is put back into the shared memory sample pool, and its state transitions from enqueued to removed. At this time, a new call to the `DataWriter`’s `get_loan()` may return the same sample.

You should not try to reuse a sample that has been written with a `DataWriter` to publish a new value. Instead, get a new sample using the `DataWriter`’s `get_loan()` and populate its content with the new value.

A sample that has not been written can be returned to the shared memory pool by using the `DataWriter`’s `discard_loan()`:

```cpp
camera_writer->discard_loan(camera_image)
```

The shared memory sample pool is destroyed when the `DataWriter` is deleted.

See the API Reference HTML documentation for more information on `get_loan()`.

### 23.1.5.1.2 Receiving data with Zero Copy transfer over shared memory

Create a `DataReader` as you normally would; see 8.3.1 Creating DataReaders on page 510.

Read the data samples:

```cpp
dds::sub::LoanedSamples<CameraImage> samples = camera_reader.take();
```

Let’s work with the first sample (assuming `samples.length() > 0` and `samples[0].info().valid()`):

```cpp
camera_image_sample = samples[0].data();
// Process the sample
process_data(camera_image_sample);
if (!camera_reader->is_data_consistent(camera_image_sample)) {
  // Sample was overwritten, ignore this sample
  rollback(camera_image_sample);
}
```

For more information on the `DataReader`’s `is_data_consistent()` API, see 23.1.5.1.3 Checking data consistency with Zero Copy transfer over shared memory below.

### 23.1.5.1.3 Checking data consistency with Zero Copy transfer over shared memory

Zero Copy transfer over shared memory makes no copies. This means the sample being processed in the subscribing application actually resides in the `DataWriter`’s send queue. The `DataWriter` in the publishing application can decide to reuse this memory to send a different sample before or while the original sample is being processed by a `DataReader`, which can lead to data consistency problems. There are several ways to prevent and detect these inconsistencies.

A reliable `DataWriter` will not attempt to reuse sample memory if the sample has not been acknowledged. With reliable communication and application-level acknowledgments (see 7.3.12 Application Acknow-
ledgment on page 317), the subscribing application can prevent the writer from reusing the sample by delaying the acknowledgment until after the sample has been processed.

**Note:** Application Acknowledgment is not currently available with RTI Connext DDS Micro.

Applications can also use other, custom, application-level mechanisms to guarantee data consistency between the publisher and the subscriber.

Without an application-level synchronization mechanism, when the application's DataWriter and DataReader are not synchronized, the subscribing application can use the DataReader's `is_data_consistent()` API to detect data inconsistencies, as long as the type is not annotated with `@language_binding` (FLAT_DATA). If the type is FlatData, reading a data sample while the DataWriter is reusing it is undefined behavior.

If the type is not FlatData, for `is_data_consistent()` to work, configure the DataWriter's 7.5.25 TRANSFER_MODE QoSPolicy on page 457 setting `writer_qos.transfer_mode shmenv_ref_settings.enable_data_consistency_check` to true (the default). A DataWriter with this setting sends a special sequence number associated with each sample as an inline QoS (metadata), which can be used to check the sample's validity at the DataReader with the DataReader's `is_data_consistent()` API. Simply, the API checks if the shared memory space has been reused for that sample. If it has, the data is inconsistent.

If data consistency checks are disabled, `is_data_consistent()` will return a PRECONDITION_NOT_MET error.

The `is_data_consistent()` API helps detect a data inconsistency, not prevent it. Therefore, the recommended way of using the API is to follow this general scheme:

```cpp
process(data);
if (!reader->is_data_consistent(data, sample_info))
    discard(processed_data);
```

When `is_data_consistent()` returns true after the sample has been processed, subscribers can be sure processed data was not inconsistent and can be trusted (e.g., by committing it to a database). When `is_data_consistent()` returns false, processed data should be discarded. If `is_data_consistent()` is only called before processing data, it could return true at that point but the sample could be modified while being processed, leading to a race condition. Therefore, if you want to call `is_data_consistent()` before processing the data (for instance, because the processing is expensive), that is fine, but be sure to also call it after processing the data.

If the publisher sends data in best-effort mode and the expected send frequency is known in advance, the DataWriter's resource limits can be configured with an appropriate `writer_loaned_sample_allocation` max count (see the API Reference HTML documentation) to minimize the chances of sample reuse and of `is_data_consistent()` returning false.

### 23.1.5.1.4 Languages Supported by Zero Copy Transfer Over Shared Memory

Zero Copy transfer over shared memory is supported in the C, Modern C++, and Traditional C++ APIs.
23.1.5.1.5 Interactions with RTI Security Plugins and Compression

When you use security in combination with Zero Copy transfer over shared memory, the samples in the shared memory segment are not serialized and are therefore not protected, regardless of the selected protection kind. With Zero Copy, Security Plugins only protects the 16-byte references sent to DataReaders. You can use any protection kind to protect the reference (see "Securing DDS Messages on the Wire" and "Understanding ProtectionKinds" in the RTI Security Plugins User's Manual).

If a DataWriter is using Zero Copy transfer over shared memory, the samples sent to DataReaders over non-shared memory transports will be serialized and protected according to the configured protection kinds (see "Related Governance Rules" in the RTI Security Plugins User's Manual).

Likewise, with compression (see 7.5.3 DATA_REPRESENTATION QosPolicy on page 380), the samples in the shared memory segment are not serialized and are therefore not compressed, regardless of the compression setting. However, the shared memory reference (see 23.1.5 Zero Copy Transfer Over Shared Memory on page 962) will be compressed if the writer_compression_threshold is set to a value less than or equal to 16 bytes. To avoid this compression when using Zero Copy, it is recommended to set the writer_compression_threshold to a value greater than 16.

If a DataWriter is using Zero Copy transfer over shared memory, the samples sent to DataReaders over non-shared memory transports will be serialized and compressed according to the compression settings.

23.1.5.1.6 Notes on Batching

A Zero Copy DataWriter (a DataWriter that sends sample references) cannot batch samples. That is, Connext DDS will not let you set up a Zero Copy DataWriter to use batching. A Zero Copy DataReader, however, can receive batched samples from a regular DataWriter as well as all samples from a Zero Copy DataWriter.

23.1.5.2 Other Considerations

23.1.5.2.1 Type Matching for Zero Copy Transfer Over Shared Memory

The default value for TypeConsistencyEnforcementQosPolicy kind is AUTO_TYPE_COERCION.

For a regular DataReader, AUTO_TYPE_COERCION is translated to ALLOW_TYPE_COERCION. A Zero Copy DataReader, however, should use a topic type that is identical to its matched Zero Copy DataWriter’s topic type, because it accesses the sample directly in the DataWriter queue. Therefore, AUTO_TYPE_COERCION for a Zero Copy DataReader is translated to DISALLOW_TYPE_COERCION. The creation of a Zero Copy DataReader with ALLOW_TYPE_COERCION will return an error.

See 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594.

23.1.5.2.2 Resource Limits Related to Zero Copy Transfer Over Shared Memory

There are resource limits on the DataWriter, DataReader, and DomainParticipant that configure different aspects of Zero Copy transfer over shared memory.
23.2 Reducing Bandwidth Usage

DataWriter Resource Limits

The `writer_loaned_sample_allocation` setting configures the initial and maximum number of loaned samples managed by the `DataWriter`. It also configures the growth policy.

By default this setting is derived from the DDS_ResourceLimitsQosPolicy: the initial and maximum counts are equal to `initial_samples + 1` and `max_samples + 1`. The incremental_count defaults to initial_count if the initial_count is not the same as max_count. If these are the same, then `incremental_count` defaults to 0.

If you want to extend the time to reuse a sample, use a large sample pool by increasing the initial_count of the `writer_loaned_sample_allocation`.

See 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 401.

DataReader Resource Limits

The `shmemp_ref_transfer_modeAttached_segment_allocation` setting configures the initial and maximum shared memory segments to which a `DataReader` can attach.

By default this setting is derived from other fields in the DDS_DataReaderResourceLimitsQosPolicy: the initial and maximum counts of shared memory segments are equal to `initial_remote_writers` and `max_remote_writers`. The incremental_count defaults to -1 (doubling of resources) if the initial_count is not the same as max_count. If these are the same, then incremental_count defaults to 0.

The `max_count` controls the maximum number of shared memory segments that a `DataReader` can attach at a time. Once this limit is hit, if there is a need to attach to a new segment, the `DataReader` will try to detach from a segment that doesn’t contain any loaned samples and attach to the new segment.

If there are samples loaned in all the attached segments, then the new segment will not be attached and this will result in losing the sample.

See 8.6.2 DATA_READERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 576.

DomainParticipant Resource Limits

The `shmemp_ref_transfer_modeMax_segments` setting sets the maximum number of shared memory segments that can be created by all `DataWriters` belonging to the participant. The default value of this setting is 500. The maximum value of this setting will be limited by the operating system setting that controls the system wide maximum number of shared memory segments.

See 9.5.4 DOMAIN_PARTICIPANTRESOURCE_LIMITS QosPolicy (DDS Extension) on page 654.

23.2 Reducing Bandwidth Usage

Another important consideration when sending large samples (besides 23.1 Reducing Latency on page 944) is bandwidth usage. Connext DDS allows you to compress samples using different builtin
23.3 Large Data Fragmentation

There are two types of fragmentation: IP-level fragmentation and DDS-level fragmentation.

IP-level fragmentation occurs when the payload provided from the transport layer (typically UDP or TCP) exceeds the maximum payload size that fits in a link frame (also known as the link maximum transmission unit, or link MTU). If the network is an Ethernet network, then the link MTU is the maximum size of an Ethernet frame. When the receiver NIC gets IP fragments, it stores them in a buffer until all the fragments are received and can be reassembled to form UDP datagrams or TCP segments. When all the fragments are received, the reassembly is performed and the message is provided to the application layer.

If you try to send a DDS sample whose size is bigger than the MTU and you have not set up DDS-level fragmentation, you will see IP-level fragmentation. IP-level fragmentation is known to be fragile and can lead to communication issues if your system is not configured properly. For example, when your application relies on the transport to fragment the data and one fragment is lost, then all of the fragments need to be resent to repair the missing fragment—whereas if you use Reliable reliability (see 7.5.21 RELIABILITY QosPolicy on page 447), Connext DDS can repair a single lost DDS fragment.

The following diagrams show the differences between IP-level fragmentation and DDS-level fragmentation. RTPS, UDP, and IP headers are not shown in the diagrams, for simplification purposes.
23.3 Large Data Fragmentation

**IP-level fragmentation scenario**

**Application**

message\_size\_max > payload > MTU

---

**RTI Connext**

no fragmentation at DDS level

---

**IP Layer**

Fragment 1
MTU bytes

Fragment 2
MTU bytes

... 

Fragment N
Payload − (N-1)*MTU bytes

No DDS fragmentation, since payload < message\_size\_max

IP-level fragmentation, since payload > MTU
The main advantages of letting DDS do the fragmentation instead of letting the IP layer do it are as follows:

- IP packets containing DATA_FRAG messages (DDS fragments) are automatically provided from the NIC’s buffer to the DDS application without having to wait for reassembly. This helps prevent overflow of the NIC’s buffer due to many fragments.
- The middleware handles fragmentation and reassembly of fragments. As a result, when using the Reliable 7.5.21 RELIABILITY QosPolicy on page 447, if an IP packet containing a DATA_FRAG is not received, Connext DDS’s reliable protocol will try to repair the missing DATA_FRAG instead of the entire DDS packet. This may help reduce network traffic in scenarios with reliable communication. It is highly recommended to use Reliable reliability in combination with fragmentation; otherwise a single lost fragment will cause the entire sample to be dropped, leading to excessive sample losses.

The main cost of using DDS-level fragmentation is that having Connext DDS handle fragmentation may introduce a performance degradation compared to an ideal case where there are no IP-level fragmentation issues. However, if there are IP-level fragmentation issues in your system, DDS-level fragmentation is a good way to avoid them. There are many different types of IP-level fragmentation issues, including, but
not limited to, mismatched MTU sizes across your network path, OS-specific implementation limitations, and hardware that simply does not allow IP fragment forwarding.

**Note:** Batching does not currently support DDS-level fragmentation (also known as RTPS fragmentation). If you use batching, you will currently not be able to take advantage of Connext DDS-level fragmentation. This means that your batch size has to be set to a value smaller than the minimum transport MTU across all the installed Connext DDS transports. (You configure the MTU by setting `message_size_max` in the transport properties. See the next section, 23.3.1 Avoiding IP-Level Fragmentation below.)

You can configure the batch size for user data using either the `max_data_bytes` or `max_samples` QoS values in the 7.5.2 BATCH QosPolicy (DDS Extension) on page 374. In either case, you need to take into account that there is some overhead of metadata per sample in a batch that can be as big as 120 bytes per sample depending on what DDS features you use. A common value when using keyed topics is 40 bytes of metadata per sample and 12 for unkeyed topics.

### 23.3.1 Avoiding IP-Level Fragmentation

IP-level fragmentation can be avoided if the DDS payload (plus UDP headers) size is shorter than the Ethernet MTU. The most common Ethernet MTU size is 1500 bytes (although this size should not be assumed, since there are many cases in which it is set to a value other than 1500). The maximum UDP payload that fits on a 1500-byte Ethernet MTU is 1472 bytes. This is because, out of the 1500 bytes in the Ethernet MTU, 20 bytes are used by the IP header and 8 more by the UDP header. You can easily know the size of your NIC’s MTU in Linux systems with the following command:

```
> ifconfig
```

In Windows systems, the MTU for your NICs is shown by this command:

```
> netsh interface ipv4 show subinterface
```

Connext DDS provides a property, `message_size_max`, to set the maximum size of an RTPS packet. See 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy on page 801 for information on how to set transport properties. Samples that have a serialized size larger than the `message_size_max` will be fragmented by DDS. Therefore, setting this property to a value less than or equal to the maximum UDP payload that fits in the Ethernet MTU (that is, smaller than 1472 bytes in the common case) makes DDS fragment the data packets so that each RTPS message can fit in a single Ethernet frame. These DDS fragments are referred to as DATA_FRAG messages.

**Note:** MTU sizes are not necessarily uniform across an entire network path from source to destination. In these cases, it is important to understand the MTU sizes throughout your network and to set the DDS `message_size_max` to a value smaller than the smallest payload that fits in the MTU size in your network. TCP avoids IP-level fragmentation and automatically detects MTU sizes across a network path through a process called Path MTU Discovery. If you’re using UDP, then it is currently up to you to know and understand the MTU sizes in your network if you want to avoid IP-level fragmentation.
A more granular configuration of DDS-level fragment management can be controlled with properties such as `max.fragments_per_sample` (see 8.6.2 `DATA_READERRESOURCE_LIMITS` QosPolicy (DDS Extension) on page 576).

While Connext DDS supports unbounded types and data fragmentation, there are practical serialization limits for any given sample. These limits are described in 3.10 Data Sample Serialization Limits on page 164.

Note: Features that are targeted at applications that handle large data, like the FlatData language binding and Zero Copy over shared memory features (see Chapter 23 Sending Large Data on page 943), have no effect on how data is fragmented by DDS.

### 23.3.2 Reliable Reliability

If you use Best Effort reliability (see 7.5.21 `RELIABILITY` QosPolicy on page 447), the application is not going to try to recover any lost DDS-level fragments, so if any fragments are lost, the DataReader will discard the entire sample. Depending on its size, the sample could have a lot of fragments, in which case the DataReader is more likely to lose a fragment (and therefore, the entire sample). By using Reliable 7.5.21 `RELIABILITY` QosPolicy on page 447, if a fragment is lost, Connext DDS will try to recover it. This is why it's usually recommended to use Reliable reliability if you are using DDS-level fragmentation.

For more information, see the 7.5.21 `RELIABILITY` QosPolicy on page 447.

### 23.3.3 Asynchronous Publishing

DDS-level fragmentation requires asynchronous publication if you are using Reliable 7.5.21 `RELIABILITY` QosPolicy on page 447. Sending reliable samples larger than the transport's `message_size_max` requires asynchronous publication so that the fragmentation process can take place outside of the context of the thread that wrote the sample.

If you're using Best Effort reliability, samples larger than the `message_size_max` will be fragmented; however, this configuration (Best Effort, plus fragmentation) is not recommended because you're more likely to drop samples. The error "CALLMSRSrWriterService_on_Submessage!write resend. Reliable large data requires asynchronous write" comes from having a serialized sample that is greater than the transport's `message_size_max` while the 7.5.21 `RELIABILITY` QosPolicy on page 447 is set to `RELIABLE_RELIABILITY_QOS` without asynchronous publishing being enabled.

To fragment DDS packets while using Reliable reliability, set `kind` in the 7.5.20 `PUBLISH_MODE` QosPolicy (DDS Extension) on page 444 to `ASYNCHRONOUS_PUBLISH_MODE_QOS`. With these settings, Connext DDS will use a separate thread to send the fragments. This will relieve your application thread from doing the fragmentation and sending work. For more information about the asynchronous publisher, see 7.4.1 `ASYNCHRONOUS_PUBLISHER` QosPolicy (DDS Extension) on page 345.

It may also be necessary to set the builtin `PublicationBuiltinTopicData` and `SubscriptionBuiltinTopicData` `DataWriters` publish mode to be asynchronous. This is done through the 9.5.3 `DISCOVERY_CONFIG` QosPolicy (DDS Extension) on page 644 (see details in 23.3.5 Example on the next page). The most
common cause of a large PublicationBuiltinTopicData or SubscriptionBuiltinTopicData sample is the serialized TypeCode or TypeObject, but you may also be sending a lot of properties (via the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439) or have a large ContentFilteredTopic filter expression, among other variably sized fields, which could be leading to larger sample sizes. It may also be the case that the samples are not particularly large, but if you have set the message_size_max to be a small value to force DDS-level fragmentation, the samples sent by the builtin DataWriters may exceed this size and require fragmentation.

For more information on TypeObjects, see the following:

- 3.1.3.1 Sending Type Information on the Network on page 38
- Type Representation, in the RTI Connext DDS Core Libraries Extensible Types Guide
- 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654

23.3.4 Flow Controllers

The asynchronous publish mode requires a FlowController. If no FlowController is defined, the default FlowController will be used. With the default FlowController, the DATA_FRAGs will be written as fast as the DataWriter can write them, which might overload the network or the DataReaders. See 7.6 FlowControllers (DDS Extension) on page 470.

An example on how to set the DataWriter to be asynchronous is shown below.

23.3.5 Example

The following example shows the QoS settings that do the following:

- Set the DataWriter to be asynchronous.
- Set the builtin DataWriters to be asynchronous.
- Enable Reliable 7.5.21 RELIABILITY QosPolicy on page 447 on the DataWriter and DataReader. DataWriters are configured as reliable by default, so this is technically not required. DataReaders are configured for best effort communication by default, so enabling reliability on the DataReader is a required step in order for the DataWriter and DataReader to communicate reliably with each other. See 7.5.21 RELIABILITY QosPolicy on page 447.
- Disable the shared memory transport (since our discussion thus far has focused on IP transports and the relationship between IP-layer fragmentation and DDS-layer fragmentation, not shared memory fragmentation).
- Set the maximum payload size for RTPS packets by configuring message_size_max.

<!-- Set the DataWriter to be asynchronous and reliable -->
<datawriter_qos>
  <publish_mode>
23.3.6 Fragmentation Statistics

You can monitor fragmented (DATA_FRAG) messages via the 7.3.6.3 DATA_WRITER_PROTOCOL_STATUS on page 298 and 8.3.7.3 DATA_READER_PROTOCOL_STATUS on page 520, which are also visible through RTI Monitor (see Part 9: RTI Monitoring Library on page 1223).
Chapter 24 Topic Queries

TopicQueries allow a DataReader to query the sample cache of its matching DataWriters. You can create a TopicQuery with the DataReader's create_topic_query() API. When a DataReader creates a TopicQuery, DDS will propagate TopicQueries to other DomainParticipants and their DataWriters. When a DataWriter matching with the DataReader that created the TopicQuery receives it, it will send the cached samples that pass the TopicQuery's filter.

Only samples that fall within the writer_depth (in the 7.5.9 DURABILITY QosPolicy on page 411) for an instance are evaluated against the TopicQuery filter. While the DataWriter is waiting for acknowledgements from one or more DataReaders, there may temporarily be more than writer_depth samples per instance in the DataWriter's queue if the 7.5.12 HISTORY QosPolicy on page 420 depth is set to a higher value than writer_depth. Those additional samples past writer_depth are not eligible to be sent in response to the TopicQuery.

To configure how to dispatch a TopicQuery, use the DataWriter's 7.5.24 TOPIC_QUERY_DISPATCH_QosPolicy (DDS Extension) on page 455. By default, a DataWriter ignores TopicQueries unless they are explicitly enabled using this policy.

The delivery of TopicQuery samples occurs in a separate RTPS channel. This allows DataReaders to receive TopicQuery samples and live samples in parallel. This is a key difference with respect to the 7.5.9 DURABILITY QosPolicy on page 411.

Late-joining DataWriters will also discover existing TopicQueries. To delete a TopicQuery you must use the DataReader's delete_topic_query().

After deleting a TopicQuery, new DataWriters will not discover it and existing DataWriters currently publishing cached samples may stop before delivering all of them.

By default, a TopicQuery queries the samples that were in the DataWriter's queue at the time the DataWriter received the TopicQuery. However, a TopicQuery can be created in “continuous” mode; in this case, a DataWriter will continue delivering samples that pass a continuous TopicQuery filter until the DataReader application explicitly deletes it.
The samples received in response to a TopicQuery are stored in the associated DataReader's cache. Any of the read/take operations can retrieve TopicQuery samples. The field DDS_SampleInfo::topic_query_guid associates each sample with its TopicQuery. If the read sample is not in response to a TopicQuery, this field will be DDS_GUID_UNKNOWN.

You can choose to read or take only TopicQuery samples, only live samples, or both. To support this, ReadConditions and QueryConditions provide the DataReader's create_querycondition_w_params() and create_readcondition_w_params() APIs.

Each TopicQuery is identified by a GUID that can be accessed using the TopicQuery's get_guid() method.

24.1 Reading TopicQuery Samples

Data samples that are received by a DataReader in response to a TopicQuery can be identified with two pieces of information from the corresponding DDS_SampleInfo to the sample. First, if the DDS_SampleInfo::topic_query_guid is not equal to DDS_GUID_UNKNOWN, then the sample is in response to the TopicQuery with that GUID. Second, if the sample is in response to a TopicQuery and the DDS_SampleInfo::flag DDS_INTERMEDIATE_TOPIC_QUERY_SAMPLE flag is set, then this is not the last sample in response to the TopicQuery for a DataWriter identified by DDS_SampleInfo::original_publication_virtual_guid. If that flag is not set, then there will be no more samples corresponding to that TopicQuery coming from the DataWriter.

24.2 Debugging Topic Queries

There are a number of ways in which to gain more insight into what is happening in an application that is creating Topic Queries.

24.2.1 The Built-in ServiceRequest DataReader

TopicQueries are communicated to publishing applications through a built-in ServiceRequest channel. The ServiceRequest channel is designed to be generic so that it can be used for many different purposes, one of which is TopicQueries.

When a DataReader creates a TopicQuery, a ServiceRequest message is sent containing the TopicQuery information. Just as there are built-in DataReaders for ParticipantBuiltInTopicData, SubscriptionBuiltInTopicData, and PublicationBuiltInTopicData, there is a fourth built-in DataReader for ServiceRequests. This built-in DataReader can be retrieved using the built-in Subscriber and its lookup_datareader(). The topic name is DDS_SERVICE_REQUEST_TOPIC_NAME. Installing a listener with the DataReaderListener's on_data_available callback() implemented will allow a publishing application to be notified whenever a TopicQuery has been received from a subscribing application.

The service_id of a ServiceRequest corresponding to a TopicQuery will be DDS_TOPIC_QUERY_SERVICE_REQUEST_ID and the instance_id will be equal to the GUID of the TopicQuery.
The **request_body** is a sequence of bytes containing more information about the TopicQuery. This information can be retrieved using the `DDS_TopicQueryHelper_topic_query_data_from_service_request()` function. The resulting TopicQueryData contains the TopicQuerySelection that the TopicQuery was created with, the GUID of the original `DataReader` that created the TopicQuery, and the topic name of that `DataReader`.

**Note:** When TopicQueries are propagated through one or more instances of Routing Service, the last `DataReader` that issued the TopicQuery will be a Routing Service `DataReader`. The `DDS_TopicQueryData::original_related_reader_guid`, however, will be that of the first `DataReader` to have created the TopicQuery.

## 24.2.2 The on_service_request_accepted() DataWriter Listener Callback

It is possible that a ServiceRequest for a TopicQuery is received but is not immediately dispatched to a `DataWriter`. This can happen, for example, if a `DataWriter` was not matching with a `DataReader` at the time that the TopicQuery was received by the publishing application. The `DDS_DataWriterListener's on_service_request_accepted()` callback notifies a `DataWriter` when a ServiceRequest has been dispatched to that `DataWriter`. The `DDS_ServiceRequestAcceptedStatus` provides information about how many ServiceRequests have been accepted by the `DataWriter` since the last time that the status was read. The status also includes the `DDS_ServiceRequestAcceptedStatus::last_request_handle`, which is the InstanceHandle of the last ServiceRequest that was accepted. This instance handle can be used to read samples per instance from the built-in ServiceRequest `DataReader` and correlate which ServiceRequests have been dispatched to which `DataWriters`.

## 24.3 System Resource Considerations

### 24.3.1 Publishing Application

On the publishing side, the resource allocation associated with TopicQueries can be controlled using **remote_topic_query_allocation** (in the 9.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 654 at the DomainParticipant level).

At the `DataWriter` level, you can control how many TopicQueries can be served in parallel by the `DataWriter` by setting the resource limit `max_active_topic_queries` in the 7.5.6 DATA_WRITERRESOURCE_LIMITS QosPolicy (DDS Extension) on page 401).

### 24.3.2 Subscribing Application

On the `DataReader` side, each TopicQuery will get its own resources. These resources will not interfere with the resource limits associated with live data samples or other TopicQueries. For example, if `max_samples` (see 7.5.22 RESOURCE_LIMITS QosPolicy on page 451) is set to 10 and the `DataReader` creates one TopicQuery, then the `DataReader` will be able to store 10 samples for that TopicQuery and 10 samples for live data.
The maximum number of active TopicQueries that can be associated with a *DataReader* is configured using the resource limit `max_topic_queries` (see 8.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) on page 576).
Chapter 25 Troubleshooting

This chapter contains tips on troubleshooting Connext DDS applications. For an up-to-date list of frequently asked questions, see the Knowledge Base on the RTI Community Portal: https://community.rti.com/kb. There you can find example code, general information on Connext DDS, performance information, troubleshooting tips, and technical details.

25.1 What Version am I Running?

There are three ways to obtain version information:

- By looking at the revision files, as described in 25.1.1 Finding Version Information in Revision Files below.
- By using Visual Studio or the command line, as described in 25.1.2 Finding Version Information on Windows or Linux Systems on the next page.
- Programmatically at run time, as described in 25.1.3 Finding Version Information Programmatically on the next page.

25.1.1 Finding Version Information in Revision Files

In the top-level directory of your Connext DDS installation ($NDDSHOME), you will find text files that include revision information. The files are named rev_<product>_rtidds.<version>. For example, you might see files called rev_host_rtidds.6.x.y and rev_persistence_rtidds.6.x.y (where x and y stand for the version numbers of the current release). Each file contains more details, such as a patch level and if the product is license managed.

For example:

Host Build 6.x.y rev 04 (0x04050200)

The revision files for Connext DDS target libraries are in the same directory as the libraries ($NDDSHOME/lib/<architecture>).
25.1.2 Finding Version Information on Windows or Linux Systems

Another way to find the version is with these commands:

- On Windows platforms, run the DUMPBIN utility that comes with Visual Studio®. (You could also use any other COFF dumper application.) For example:

  ```
  DUMPBIN/HEADERS nddcore.dll
  ```

  You will find the version number encoded in the 'image version' line in the 'OPTIONAL HEADER VALUES' section:

  ```
  OPTIONAL HEADER VALUES
  <snip>
  50200.00 image version
  <snip>
  ```

  The format is <major_version><minor_version><terciary_version><patch_version>. For example, version 5.2.6.3 would appear as image version 50206.03.

- On Linux platforms, run the command `strings` on the library in question and filter for 'BUILD'. For example:

  ```
  strings libnddsc.so | grep BUILD
  ```

  You will see a string similar to

  ```
  NDDSCORE_VERSION_5.2.6.0_BUILD_2017-01-27T15:43:23-08:00_RTI_RELEASE
  ```

25.1.3 Finding Version Information Programmatically

The methods in the NDDSConfigVersion class can be used to retrieve version information for the Connext DDS product, the core library, and the C, C++ or Java libraries.

The version information includes four fields:

- A major version number
- A minor version number
- A release number
- A build number

Table 25.1 NDDSConfigVersion Operations lists the available operations (they will vary somewhat depending on the programming language you are using; consult the API Reference HTML documentation for more information).
Table 25.1 NDDSConfigVersion Operations

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>To retrieve version information in a structured format</td>
<td>get_product_version</td>
<td>Gets version information for the Connext DDS product.</td>
</tr>
<tr>
<td></td>
<td>get_core_version</td>
<td>Gets version information for the Connext DDS core library.</td>
</tr>
<tr>
<td></td>
<td>get_c_api_version</td>
<td>Gets version information for the Connext DDSC library.</td>
</tr>
<tr>
<td></td>
<td>get_cpp_api_version</td>
<td>Gets version information for the Connext DDSC++ library.</td>
</tr>
<tr>
<td>To retrieve version information in string format</td>
<td>to_string</td>
<td>Converts the version information for each library into a string. The strings for each library are put in a single hyphen-delimited list.</td>
</tr>
</tbody>
</table>

The `get_product_version()` operation returns a reference to a structure of type `DDS_ProductVersion_t`:

```c
struct NDDS_Config_ProductVersion_t {
    DDS_Char major;
    DDS_Char minor;
    DDS_Char release;
    DDS_Char revision;
};
```

The other `get_*_version()` operations return a reference to a structure of type `NDDS_Config_LibraryVersion_t`:

```c
struct NDDS_Config_LibraryVersion_t {
    DDS_Long major;
    DDS_Long minor;
    char release;
    DDS_Long build;
};
```

The `to_string()` operation returns version information for the Connext DDS core, followed by the C and C++ API libraries, separated by hyphens. For example:
25.2 Controlling Messages from Connext DDS

Connext DDS's built-in logging system provides several types of messages to help you debug your system and alert you to errors during run time. You can control how much information is reported and where it is logged. By default, the built-in logging system writes to the standard output, but you can configure it to use a logging file or an output device such as a custom logging device or the Distributed Logger. (See Part 10: RTI Distributed Logger on page 1239.) See also Table 25.4 NDDSConfigLogger Operations and 25.2.2 Configuring Logging via XML on page 992 for information on configuring the built-in logging system.

How much information is logged is known as the verbosity setting. Table 25.2 Message Logging Verbosity Levels describes the increasing verbosity levels. Note that the verbosities are cumulative: logging at a high verbosity means also logging all lower verbosity messages. If you change nothing, the default verbosity will be set to NDDS_CONFIG_LOG_VERBOSITY_ERROR.

Logging at high verbosities can be detrimental to your application's performance. You should generally not set the verbosity above NDDS_CONFIG_LOG_VERBOSITY_WARNING, unless you are debugging a specific problem.

### Table 25.2 Message Logging Verbosity Levels

<table>
<thead>
<tr>
<th>Verbosity (NDDS_CONFIG_LOG_VERBOSITY_*)</th>
<th>Description</th>
<th>Log level values corresponding to this verbosity (NDDS_CONFIG_LOG_LEVEL_*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SILENT</td>
<td>No messages will be logged. (lowest verbosity)</td>
<td>-</td>
</tr>
</tbody>
</table>
25.2 Controlling Messages from Connext DDS

### Table 25.2 Message Logging Verbosity Levels

<table>
<thead>
<tr>
<th>Verbosity (NDDS_CONFIG_LOG_VERBOSITY_*)</th>
<th>Description</th>
<th>Log level values corresponding to this verbosity (NDDS_CONFIG_LOG_LEVEL_*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERROR (default level for all categories)</td>
<td>Log only high-priority error messages. An error indicates something is wrong with how Connext DDS is functioning. The most common cause of this type of error is an incorrect configuration.</td>
<td>ERROR, FATAL_ERROR</td>
</tr>
<tr>
<td>WARNING</td>
<td>Additionally log warning messages. A warning indicates that Connext DDS is taking an action that may or may not be what you intended. Some configuration information is also logged at this verbosity to aid in debugging.</td>
<td>WARNING, ERROR, FATAL_ERROR</td>
</tr>
<tr>
<td>STATUS_LOCAL</td>
<td>Additionally log verbose information about the lifecycles of local Connext DDS objects.</td>
<td>STATUS_LOCAL, WARNING, ERROR, FATAL_ERROR</td>
</tr>
<tr>
<td>STATUS_REMOTE</td>
<td>Additionally log verbose information about the lifecycles of remote Connext DDS objects.</td>
<td>STATUS_REMOTE, STATUS_LOCAL, WARNING, ERROR, FATAL_ERROR</td>
</tr>
<tr>
<td>STATUS_ALL</td>
<td>Additionally log verbose information about periodic activities and Connext DDS threads. (highest verbosity)</td>
<td>DEBUG, STATUS_REMOTE, STATUS_LOCAL, WARNING, ERROR, FATAL_ERROR</td>
</tr>
</tbody>
</table>

You will typically change the verbosity of all of Connext DDS at once. However, in the event that such a strategy produces too much output, you can further discriminate among the messages you would like to see. The types of messages logged by Connext DDS fall into the categories listed in Table 25.3 Message Logging Categories; each category can be set to a different verbosity level.

### Table 25.3 Message Logging Categories

<table>
<thead>
<tr>
<th>Category (NDDS_CONFIG_LOG_CATEGORY_*)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLATFORM</td>
<td>Messages about the underlying platform (hardware and OS).</td>
</tr>
<tr>
<td>COMMUNICATION</td>
<td>Messages about data serialization and deserialization and network traffic.</td>
</tr>
<tr>
<td>DATABASE</td>
<td>Messages about the internal database of Connext DDS objects.</td>
</tr>
<tr>
<td>ENTITIES</td>
<td>Messages about local and remote entities and the discovery process.</td>
</tr>
<tr>
<td>API</td>
<td>Messages about Connext DDS's API layer, such as method argument validation and what QoS is being used (for details on QoS information, see 19.3.3.5 Viewing Resolved QoS Values on page 869).</td>
</tr>
<tr>
<td>ALL</td>
<td>Messages about all of the categories (default value)</td>
</tr>
</tbody>
</table>
The methods in the NDDSConfigLogger class can be used to change verbosity settings, as well as the destination and format of the logged messages. Table 25.4 NDDSConfigLogger Operations lists the available operations; consult the API Reference HTML documentation for more information.

### Table 25.4 NDDSConfigLogger Operations

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Verbosity for all Categories</td>
<td>get_verbosity</td>
<td>Gets the current verbosity. If per-category verbosities are used, returns the highest verbosity of any category.</td>
</tr>
<tr>
<td></td>
<td>set_verbosity</td>
<td>Sets the verbosity of all categories.</td>
</tr>
<tr>
<td>Change Verbosity for a Specific Category</td>
<td>get_verbosity_by_category</td>
<td>Gets/Sets the verbosity for a specific category.</td>
</tr>
<tr>
<td></td>
<td>set_verbosity_by_category</td>
<td></td>
</tr>
<tr>
<td>Change Destination of Logged Messages</td>
<td>get_output_file</td>
<td>Returns the file to which messages are being logged, or NULL for the default destination (standard output on most platforms).</td>
</tr>
<tr>
<td></td>
<td>set_output_file</td>
<td>Redirects future logged messages to a set of files. For better performance when log messages are generated frequently, the log messages are not flushed into a file immediately after they are generated. In other words, while writing a log message, Connext DDS only calls the function fwrite(); it does not call the function fflush(). If your application requires a different flushing behavior, you may configure a custom logging device (see 25.2.3 Customizing the Handling of Generated Log Messages on page 993).</td>
</tr>
<tr>
<td></td>
<td>get_output_device</td>
<td>Returns the logging device installed with the logger.</td>
</tr>
<tr>
<td></td>
<td>set_output_device</td>
<td>Registers a specified logging device with the logger. See 25.2.3 Customizing the Handling of Generated Log Messages on page 993</td>
</tr>
<tr>
<td>Change Message Format</td>
<td>get_print_format</td>
<td>Gets/Sets the current message format for the log level NDDS_CONFIG_LOG_LEVEL_ERROR. See 25.2.1 Format of Logged Messages on the next page.</td>
</tr>
<tr>
<td></td>
<td>set_print_format</td>
<td>Use get_print_format_by_log_level and set_print_format_by_log_level to retrieve/set the format for other log levels.</td>
</tr>
<tr>
<td></td>
<td>get_print_format_by_log_level</td>
<td></td>
</tr>
<tr>
<td></td>
<td>set_print_format_by_log_level</td>
<td>Gets/Sets the current message format, by log level, that Connext DDS is using to log diagnostic information.</td>
</tr>
</tbody>
</table>
25.2.1 Format of Logged Messages

You can control the amount of information in each message with the `set_print_format()` or `set_print_format_by_log_level()` operation (see Table 25.4 NDDSConfigLogger Operations). The format options are listed in Table 25.5 Message Formats.

<table>
<thead>
<tr>
<th>Message Format (NDDS_CONFIG_LOG_PRINT_FORMAT_*)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFAULT</td>
<td>(default) Message, method name, and activity context (what was happening when the event occurred).</td>
</tr>
<tr>
<td>TIMESTAMPED</td>
<td>Message, method name, activity context, and timestamp.</td>
</tr>
<tr>
<td>VERBOSE</td>
<td>Message with all available context information (includes thread identifier, activity context).</td>
</tr>
<tr>
<td>VERBOSE_TIMESTAMPED</td>
<td>Message with all available context information and timestamp.</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Information (including activity context and backtrace information) for internal debugging by RTI personnel.</td>
</tr>
<tr>
<td>MINIMAL</td>
<td>Message number, method name.</td>
</tr>
<tr>
<td>MAXIMAL</td>
<td>All available fields (including backtrace information).</td>
</tr>
</tbody>
</table>

See also 25.2.1.3 Activity Context on the next page and 25.5 Logging a Backtrace for Failures on page 996.

By default, NDDS_CONFIG_LOG_PRINT_FORMAT_DEFAULT is assigned to all log levels except FATAL_ERROR. By default, FATAL_ERROR is assigned to NDDS_CONFIG_LOG_PRINT_FORMAT_DEBUG, which prints the backtrace information. See Table 25.2 Message Logging Verbosity Levels.

You could use a less verbose `print_format`, such as NDDS_CONFIG_LOG_PRINT_FORMAT_MINIMAL, for warnings, as follows:

```c
NDDS_Config_Locker *logger = NDDS_Config_Locker_get_instance();
NDDS_Config_Locker_set_print_format_by_log_level(  
    logger,  
    NDDS_CONFIG_LOG_PRINT_FORMAT_MINIMAL,  
    NDDS_CONFIG_LOG_LEVEL_WARNING));
```

You could use a more verbose `print_format`, such as NDDS_CONFIG_LOG_PRINT_FORMAT_DEBUG (which contains the backtrace) when you are troubleshooting errors, as follows:

```c
NDDS_Config_Locker *logger = NDDS_Config_Locker_get_instance();
NDDS_Config_Locker_set_print_format_by_log_level(  
    logger,  
    NDDS_CONFIG_LOG_PRINT_FORMAT_DEBUG,  
    NDDS_CONFIG_LOG_LEVEL_ERROR));
```
This way, you will reduce the amount of logging on warnings, and errors will contain more information. This configuration is key to understanding and solving issues when needed.

Of course, you are not likely to recognize all of the method names; many of the operations that perform logging are deep within the implementation of Connext DDS. However, in case of errors, logging will typically take place at several points within the call stack; the output thus implies the stack trace at the time the error occurred. You may only recognize the name of the operation that was the last to log its message (i.e., the function that called all the others); however, the entire stack trace is extremely useful to RTI support personnel in the event that you require assistance.

You may notice that many of the logged messages begin with an exclamation point character. This convention indicates an error and is intended to be reminiscent of the negation operator in many programming languages. For example, the message “!create socket” means “cannot create socket.”

### 25.2.1.1 Timestamps

Reported times are in seconds from a system-dependent starting time; these are equivalent to the output format from Connext DDS. The timestamp is in the form YYYY-MM-DD HH:MM:SS.<microseconds>, where SS is the number of seconds and <microseconds> is a fraction of that second expressed in microseconds. Enabling timestamps will result in some additional overhead for clock access for every message that is logged.

Logging of timestamps is not enabled by default. To enable it, use NDDS_Config_Logger method set_print_format().

### 25.2.1.2 Thread identification

Thread identification strings uniquely identify active threads when a message is output to the console. A thread may be a user (application) thread or one of several types of internal threads. See Chapter 21 Connext DDS Threading Model on page 908.

Logging of thread IDs is not enabled by default. To enable it, use NDDS_Config_Logger method set_print_format(). It adds the thread name to the log message, so you know which thread is responsible for the message. See 21.9 Identifying Threads Used by Connext DDS on page 916.

### 25.2.1.3 Activity Context

Many middleware APIs now store information in thread-specific storage about the activity context operation.

Activity Context provides more context about a logging message. It is a group of resources and activities associated with an action, such as the creation of an entity:

- A resource is an abstraction of an entity. It can contain attributes such as Topic or Domain ID.
- An activity is a general task that a resource is doing, such as "Getting QoS."
The activity context is one of the **NDDS_Config_LogPrintFormat** DDS logging infrastructure formats. If a format that prints activity context is selected (see Table 25.5 Message Formats), then every time **Connext DDS** logs a message, it will contain the contextual information.

For example, in the creation of a *DataWriter*, the activity context will provide information about:

- **Resource**: the *Publisher* creating the *DataWriter*. The attributes of the publisher will be GUID, *Entity* kind, name, and Domain ID.
- **Activity**: entity creation. It will have two parameters, the *Entity* kind and the *Topic*—in the example below, "Writer" and "TestTopic."

The string representation of the above activity context would be:

```
[0X101A76B,0X79E5D71,0X50EE914:0X80000008{E=Pu,N=TestPublisher,D=1}|CREATE Writer WITH TOPIC TestTopic]
```

In this example, the activity context fields are as follows:

- **GUID** is **0X101A76B,0X79E5D71,0X50EE914:0X80000003**

```
[0X101A76B,0X79E5D71,0X50EE914:0X80000003|E=Pu,N=TestPublisher,D=1]|CREATE Writer WITH TOPIC TestTopic]
```

- **Entity** name is **N=TestPublisher**

```
[0X101A76B,0X79E5D71,0X50EE914:0X80000003|E=Pu,N=TestPublisher,D=1]|CREATE Writer WITH TOPIC TestTopic]
```

- **Entity** kind is **E=Pu** (for Publisher)

```
[0X101A76B,0X79E5D71,0X50EE914:0X80000003|E=Pu,N=TestPublisher,D=1]|CREATE Writer WITH TOPIC TestTopic]
```

- **Domain ID** is **D=1**

```
[0X101A76B,0X79E5D71,0X50EE914:0X80000003|E=Pu,N=TestPublisher,D=1]|CREATE Writer WITH TOPIC TestTopic]
```

- **Activity** is **CREATE Writer WITH TOPIC TestTopic**

```
[0X101A76B,0X79E5D71,0X50EE914:0X80000003|E=Pu,N=TestPublisher,D=1]|CREATE Writer WITH TOPIC TestTopic]
```

When a *DataWriter* writes a sample, the activity context will provide information about:

- **Resource**: the *DataWriter* writing the sample. The attributes of the *DataWriter* will be GUID, name, *Entity* kind, *Topic*, data type, and Domain ID.
- **Activity**: the writing of a sample.

The string representation of this activity context would be:
25.2.1 Format of Logged Messages

In this example, the additional activity context fields are as follows:

- Topic is \texttt{T=test}

- Data type is \texttt{C=Foo}

25.2.1.4 Activity Context Strings and Attributes

The resources of the activity context can have multiple associated attributes. Those attributes provide extra information about the entity such as GUID prefix, \textit{Topic}, data type (class), \textit{Entity} kind, \textit{Entity} name, and Domain ID. The following tables describe how those attributes are represented.

### Table 25.6 Activity Context Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GUID</td>
<td>(given at the beginning of the context, such as: \texttt{0X101A76B,0X79E5D71,0X50EE914:0X1C1:0X80000003})</td>
</tr>
<tr>
<td>N</td>
<td>Name of the entity, such as \texttt{TestPublisher}</td>
</tr>
<tr>
<td>E</td>
<td>Entity kind, such as \texttt{Pu} for Publisher</td>
</tr>
<tr>
<td>D</td>
<td>Domain ID</td>
</tr>
<tr>
<td>T</td>
<td>Topic name</td>
</tr>
<tr>
<td>C</td>
<td>Data type</td>
</tr>
</tbody>
</table>

### Table 25.7 Activity Context Resources and Entities

<table>
<thead>
<tr>
<th>Entity Kind</th>
<th>Entity Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DP</td>
<td>DDS_DomainParticipant</td>
</tr>
<tr>
<td>Pu</td>
<td>DDS_Publisher</td>
</tr>
<tr>
<td>Su</td>
<td>DDS_Subscriber</td>
</tr>
<tr>
<td>To</td>
<td>DDS_TOPIC</td>
</tr>
<tr>
<td>DW</td>
<td>DDS_\langle\rangleDataWriter</td>
</tr>
<tr>
<td>DR</td>
<td>DDS_\langle\rangleDataReader</td>
</tr>
</tbody>
</table>
Table 25.8 Activity Context Activities

<table>
<thead>
<tr>
<th>String</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Entity operations:</strong></td>
<td></td>
</tr>
<tr>
<td>ENABLE</td>
<td>Entity::enable</td>
</tr>
<tr>
<td>GET_QOS</td>
<td>Entity::get_qos</td>
</tr>
<tr>
<td>SET_QOS</td>
<td>Entity::set_qos</td>
</tr>
<tr>
<td>GET_LISTENER</td>
<td>Entity::get_listener</td>
</tr>
<tr>
<td>SET_LISTENER</td>
<td>Entity::set_listener</td>
</tr>
<tr>
<td><strong>Factory operations</strong></td>
<td></td>
</tr>
<tr>
<td>CREATE &lt;Entity&gt;</td>
<td>Factory::create_&lt;entity&gt;</td>
</tr>
<tr>
<td>DELETE &lt;Entity&gt;</td>
<td>Factory::delete_&lt;entity&gt;</td>
</tr>
<tr>
<td>GET_DEFAULT_QOS &lt;Entity&gt;</td>
<td>Factory::get_default_&lt;entity&gt;._qos</td>
</tr>
<tr>
<td>SET_DEFAULT_QOS &lt;Entity&gt;</td>
<td>Factory::set_default_&lt;entity&gt;._qos</td>
</tr>
<tr>
<td><strong>Participant-specific operations:</strong></td>
<td></td>
</tr>
<tr>
<td>GET_PUBS</td>
<td>Participant::get_publishers</td>
</tr>
<tr>
<td>GET_SUBS</td>
<td>Participant::get_subscribers</td>
</tr>
<tr>
<td>LOOKUP Topic(&lt;name&gt;)</td>
<td>Participant::lookup_topicdescription</td>
</tr>
<tr>
<td>LOOKUP FlowController(&lt;name&gt;)</td>
<td>Participant::lookup_flowcontroller</td>
</tr>
<tr>
<td>IGNORE &lt;Entity&gt;(&lt;host ID&gt;)</td>
<td>Participant::ignore_&lt;entity&gt;</td>
</tr>
</tbody>
</table>

25.2.1.5 Configuring Activity Context Attributes

The attributes that `NDDS_Config_ActivityContextAttribute` uses in the string representation of the activity context can be configured through a mask. This mask indicates what resource attributes are used when Connext DDS logs a message or when the Heap Monitoring utility saves statistics for a memory allocation.

```c
void NDDS_Config_ActivityContext_set_attribute_mask(
    NDDS_Config_ActivityContextAttributeKindMask attribute_mask);
```

enum NDDS_Config_ActivityContextAttributeKind {
    NDDS_CONFIG_ACTIVITY_CONTEXT_ATTRIBUTE_GUID_PREFIX,
    NDDS_CONFIG_ACTIVITY_CONTEXT_ATTRIBUTE_TOPIC,
    NDDS_CONFIG_ACTIVITY_CONTEXT_ATTRIBUTE_TYPE,
    NDDS_CONFIG_ACTIVITY_CONTEXT_ATTRIBUTE_ENTITY_KIND,
    NDDS_CONFIG_ACTIVITY_CONTEXT_ATTRIBUTE_DOMAIN_ID,
    NDDS_CONFIG_ACTIVITY_CONTEXT_ATTRIBUTE_ENTITY_NAME
}
25.2.2 Configuring Logging via XML

Logging can also be configured using the DomainParticipantFactory’s 9.4.1 LOGGING QosPolicy (DDS Extension) on page 632 with the tags, <participant_factory_qos><logging>. The fields in the LoggingQosPolicy are described in XML using a 1-to-1 mapping with the equivalent C representation shown below:

```c
struct DDS_LoggingQosPolicy {
    NDDS_Config_LogVerbosity verbosity;
    NDDS_Config_LogCategory category;
    NDDS_Config_LogPrintFormat print_format;
    char * output_file;
};
```

The equivalent representation in XML:

```xml
<participant_factory_qos>
    <logging>
        <verbosity></verbosity>
        <category></category>
        <print_format></print_format>
        <output_file></output_file>
    </logging>
</participant_factory_qos>
```

The attribute `<is_default_participant_factory_profile>` can be set to true for the `<qos_profile>` tag to indicate from which profile to use `<participant_factory_qos>`. If multiple QoS profiles have `<is_default_participant_factory_profile>` set to true, the last profile with `<is_default_participant_factory_profile>` set to true will be used.

If none of the profiles have set `<is_default_participant_factory_profile>` to true, the profile with `<is_default_qos>` set to true will be used.

In the following example, DefaultProfile2 will be used:

```xml
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="../xsd/rti_dds_qos_profiles.xsd">
    <!-- Qos Library -->
    <qos_library name="DefaultLibrary">
        <qos_profile name="DefaultProfile1"
                 is_default_participant_factory_profile ="true">
            <participant_factory_qos>
                <logging>
                    <verbosity>ALL</verbosity>
                    <category>ENTITIES</category>
                    <print_format>MAXIMAL</print_format>
                    <output_file>LoggerOutput1.txt</output_file>
                </logging>
            </participant_factory_qos>
        </qos_profile>
    </qos_library>
</dds>
```
Note: The LoggingQosPolicy is currently the only QoS policy that can be configured using the `<participant_factory_qos>` tag.

25.2.3 Customizing the Handling of Generated Log Messages

By default, the log messages generated by Connext DDS are sent to the standard output. You can redirect the log messages to a file by using the `set_output_file()` operation,

To further customize the management of the generated log messages, you can use the Logger’s `set_output_device()` operation to install a user-defined logging device. The logging device must implement an interface with two operations: `write()` and `close()`.

Connext DDS will call the `write()` operation to write a new log message to the input device. The log message provides the text and the verbosity corresponding to the message.

Connext DDS will call the `close()` operation when the logging device is uninstalled.

**Note:** It is not safe to make any calls to the Connext DDS core library including calls to `DDS_DomainParticipant_get_current_time()` from any of the logging device operations.

For additional details on user-defined logging devices, see the API Reference HTML documentation (under Modules, RTI Connext DDS API Reference, Configuration Utilities).
25.3 Monitoring Native Heap Memory Usage

Connext DDS allows you to monitor the memory allocations done by the middleware on the native heap. This feature can be used to analyze and debug unexpected memory growth.

This feature includes the following APIs (available in all languages):

- NDDSUtilityHeapMonitoring::enable
- NDDSUtilityHeapMonitoring::disable
- NDDSUtilityHeapMonitoring::pause
- NDDSUtilityHeapMonitoring::resume
- NDDSUtilityHeapMonitoring::take_heap_snapshot

After NDDSUtilityHeapMonitoring::enable is called, you may invoke NDDSUtilityHeapMonitoring::take_heap_snapshot to save the current heap memory usage to a file. By comparing two snapshots, you can tell if new memory has been allocated and, in many cases, where.

For more information, see the API Reference HTML documentation.

25.4 Network Capture

Connext DDS allows you to capture network traffic that one or more DomainParticipants send or receive. This feature can be used to analyze and debug communication problems between your DDS applications. When network capture is enabled, each DomainParticipant will generate a pcap-based file that can then be opened by a packet analyzer like Wireshark, provided the right dissectors are installed.

To some extent, network capture can be used as an alternative to existing pcap-based network capture software (such as Wireshark). This will be the case when you are only interested in analyzing the traffic a DomainParticipant sends/receives. In this scenario, network capture will actually have some advantages over more general pcap-based network capture applications: RTI's network capture includes additional information, such as security-related data; it also removes information that is not needed, such as user data, when you want to reduce the capture size. That said, RTI's network capture is not a replacement for other pcap-based network capture applications: it only captures the traffic exchanged by the DomainParticipants, but it does not capture any other traffic exchanged through the system network interfaces.

To capture network traffic, NDDSUtilityNetworkCapture::enable must be invoked before creating any DomainParticipant. Similarly, NDDSUtilityNetworkCapture::disable must be called after deleting all participants. In between these calls, you may start, stop, pause, or resume capturing traffic for one or all DomainParticipants.

This feature includes the following APIs (available in all languages):
Capturing Shared Memory Traffic

For more information, see the API Reference HTML documentation.

25.4.1 Capturing Shared Memory Traffic

Every RTPS frame in network capture has a source and a destination associated with it. In the case of shared memory traffic, a process identifier and a port determine the source and destination endpoints.

Access to the process identifier (PID) of the source for inbound traffic requires changes in the shared memory segments. These changes would break shared memory compatibility with versions of Connext DDS earlier than 6.1.0. For this reason, by default, network capture will not populate the value of the source PID for inbound shared memory traffic.

If interoperability with pre-6.1.0 versions of Connext DDS is not necessary, you can generate capture files containing the source PID for inbound traffic. To do so, configure the value of the `dds.transport.minimum_compatibility_version` property to 6.1.0. (See 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.)

```xml
<domain_participant_qos>
  <property>
    <value>
      <element>
        <name>dds.transport.minimum_compatibility_version</name>
        <value>6.1.0</value>
        <propagate>false</propagate>
      </element>
    </value>
  </property>
</domain_participant_qos>
```

This property is never propagated, so it must be consistently configured throughout the whole system.

**Note:** Changing the value of this property affects the type of shared memory segments that Connext DDS uses. For that reason, you may see the following warning, resulting from leftover shared memory segments:

```
[0xC733A001,0xB248F671,0xAEC4A0C1:0x000001C1{D=200}|CREATE DP|ENABLE] NDDS_Transport_Shmem_is_segment_compatible:incompatible shared memory protocol detected.
Current version 4.0 not compatible with 2.0.
```
25.5 Logging a Backtrace for Failures

The leftover shared memory segments can be removed using the `ipcrm` command. See [https://community.rti.com/kb/what-are-possible-solutions-common-shared-memory-issues](https://community.rti.com/kb/what-are-possible-solutions-common-shared-memory-issues) for more information.

### 25.5 Logging a Backtrace for Failures

In some scenarios, it might be desirable to log the backtrace from the code. A backtrace is a list of the function calls that are currently active in a thread. You can usually inspect a backtrace by using debugging utilities like `gdb`, but sometimes these are not available.

Now, Connext DDS logs the backtrace when a precondition fails in debug mode and when a segmentation fault occurs, for macOS, Windows, and Linux systems. The backtrace feature is automatically enabled upon creation of the first `DomainParticipant`. (That is, you will not see the backtrace log in a failure until the first `DomainParticipant` is created.)

- Normally when a precondition fails, the execution continues and there is no information about the problem, but Connext DDS provides a backtrace with context about where the issue was.
- When a segmentation fault occurs, the processor or operating system does not always provide a core dump, but Connext DDS provides a backtrace with context about where the issue was.

For Linux systems, the output of the backtrace will look like this:

```
#1 RTIosapiProcessTester_testPrintBacktrace /connextdds/osapi.1.0/srcC/process/test/processTester.c:638 [0x417371]
#2 RTITestSetting_runTestsExt /connextdds/test.1.0/srcC/setting/Setting.c:719 [0x4623B8]
#3 RTITestSetting_runTests /connextdds/test.1.0/srcC/setting/Setting.c:905 [0x462B85]
#4 RTIosapiProcessTester_run /connextdds/osapi.1.0/srcC/process/test/processTester.c:683 [0x41750C]
#5 RTITestSetting_runTestsExt /connextdds/test.1.0/srcC/setting/Setting.c:719 [0x4623B8]
#6 RTITestSetting_runTests /connextdds/test.1.0/srcC/setting/Setting.c:905 [0x462B85]
#7 RTIosapiTester_run /connextdds/osapi.1.0/srcC/test/Tester.c:213 [0x403759] #9 ?? ??:0 [0xE8434830] #10 _start ??:?? [0x403759]
```

See the [RTI Connext DDS Core Libraries Platform Notes](https://community.rti.com/kb/rti-connext-dds-platform-notes) for further details on enabling this feature on macOS, Windows, and Linux systems.

The backtrace feature is smart enough to log the backtrace only once for a given error and not for the following errors in the same code path of the caller's functions. For example, in the failure of the creation of the `DDSDomainParticipant`, Connext DDS logs the backtrace for just one error instead of logging it for all of the error messages in the same code path:

```
U00007f86a87df700 Mx08:UDpv4SocketFactory.c:685:RTI0x2080010:invalid port 5562900
Backtrace:
#3 NDDS_Transport_UDPv4_Socket_bind_with_ip ??:?? [0xCB235C]
#4 NDDS_Transport_UDPv4_SocketFactory_create_receive_socket ??:?? [0xCB2619]
#5 NDDS_Transport_UDP_create_recvresource_rrEA_Udp.c:?? [0xCB170]
#6 RTINetioReceiver_addEntryport ??:?? [0xCA33F3]
#7 COMMENDActiveFacade_addEntryport ActiveFacade.c:?? [0xC12B56]
```
25.6 Setting Warnings for Operation Delays

You can configure logging a warning when a specific operation takes more time than expected. This capability is useful for discovering problems related to contentions, delays, and blocks. By default, these warnings are not logged. You have to explicitly enter a timeout period for the desired warning to see it. You can set these timeouts via properties in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439.

The operations for which you can set timeouts are:

- Send operation: Print a warning message when the send operation time exceeds the time threshold configured by the property `dds.participant.logging.time_based_logging.send.timeout`. The output message will look like this:

```
By default, the print_format NDDS_CONFIG_LOG_PRINT_FORMAT_DEBUG is set for the log level NDDS_CONFIG_LOG_LEVEL_FATAL_ERROR. This means that by default the backtrace is logged for precondition and segmentation faults; however, you can disable the backtrace for NDDS_CONFIG_LOG_LEVEL_FATAL_ERROR. In the following code, the log level NDDS_CONFIG_LOG_LEVEL_FATAL_ERROR uses the print_format NDDS_CONFIG_LOG_PRINT_FORMAT_DEFAULT, which does not contain the backtrace information:

NDDS_Config_Logger *logger = NDDS_Config_Logger_get_instance();
NDDS_Config_Logger_set_print_format_by_log_level(
    logger,
    NDDS_CONFIG_LOG_PRINT_FORMAT_DEFAULT,
    NDDS_CONFIG_LOG_LEVEL_FATAL_ERROR));
```

See 25.2.1 Format of Logged Messages on page 987 and 25.2 Controlling Messages from Connext DDS on page 984.
It took '0.359548' seconds to send '96' bytes to 'shmem://903A:C1C4:91F3:A80F:CA5B:3B82:0000:0000:7661', which exceeds the time threshold configured in 'dds.participant.logging.time_based_logging.send.timeout'."

- Event operations: Print a warning message when the event start or execution time exceeds the time threshold configured by the property `dds.participant.logging.time_based_logging.event.timeout`. The output message will look like this:

  The event thread took '2.502871' seconds waiting to trigger the event, which exceeds the time threshold configured in 'dds.participant.logging.time_based_logging.event.timeout'.

  The event thread took '1.436886' seconds executing the event, which exceeds the time threshold configured in 'dds.participant.logging.time_based_logging.event.timeout'.

- Process received data operation: Print a warning message when the processing of a received message on a specific port exceeds a time threshold set in `dds.participant.logging.time_based_logging.process_received_message.timeout`. The tracking ports can be configured using the property `dds.participant.logging.time_based_logging.process_received_message.tracked_ports`. The ports will be separated by a comma ','. They can be described using a regular expression, such as, "76*,1234". If ports are not specified, all of the ports will be tracked.

  The output message will look like this:

  It took '0.003795' seconds to process the received message of '496' bytes by the port '7662', which exceeds the threshold configured in 'dds.participant.logging.time_based_logging.process_received_message.timeout'."

- Authentication process (if using RTI Security Plugins): Print a warning message when the authentication operation time exceeds the time threshold configured by the property `dds.participant.logging.time_based_logging.authentication.timeout`. The output message will look like this:

  [0xC0A87A01,0x00007BFC,0x00000001:0x000201C4{E=DR,I=21}|RECEIVE FROM 0xC0A87A01,0x00007BFC,0x00000002:0x000201C3]PRESParticipant_processHandshake:It took '1.096696' seconds to authenticate the remote participant [0xC0A87A01,0x00007BFC,0x00000002], which exceeds the threshold configured in 'dds.participant.logging.time_based_logging.authentication.timeout'.

These and other properties are documented in the Property Reference Guide.
Part 4: Request-Reply and RPC Communication Patterns

As real-time and embedded applications become more complex, and require integration with enterprise applications, you may need additional communication patterns besides publish-subscribe. Perhaps your application needs certain information only occasionally—such as changes in temperature over the past hour, or even just once, such as application configuration data that is required only at startup. To get information only when needed, Connext DDS supports a request-reply communication pattern, which is described in the following sections:

- Introduction to the Request-Reply Communication Pattern (Chapter 26 on page 1000)
- Using the Request-Reply Communication Pattern (Chapter 27 on page 1006)
- Remote Procedure Calls (RPC)—Experimental Feature (Chapter 28 on page 1024)
Chapter 26 Introduction to the Request-Reply Communication Pattern

The fundamental communication pattern provided by Connext DDS is known as DDS data-centric publish-subscribe. The data-centric publish-subscribe pattern is particularly well-suited in situations where the same data must flow from one producer to many consumers, or when data is streaming continuously from producers to consumers. For example, the values produced by a temperature sensor may be observed by multiple applications, such as control applications, UI applications, supervisory applications, historians, etc.

Figure 26.1: Publish-Subscribe Overview
The publish-subscribe pattern supports multicast, which allows efficient distribution from a single source to multiple applications, devices, or subscribers simultaneously. But even with a single subscriber, the publish-subscribe pattern is still advantageous, because the publisher can push new updates to a subscriber as soon as they happen. That way the subscriber always has access to the latest data, with minimum delays, and without incurring the overhead of periodically polling what may be stale data. This efficient, low-latency access to the most current information is important for real-time applications.

### 26.1 The Request-Reply Pattern

As applications become more complex, it often becomes necessary to use other communication patterns in addition to publish-subscribe. Sometimes an application needs to get a one-time snapshot of information; for example, to make a query into a database or retrieve configuration parameters that never change. Other times an application needs to ask a remote application to perform an action on its behalf; for example, to invoke a remote procedure call or a service.

To support these scenarios, Connext DDS includes support for the request-reply communication pattern.

**Figure 26.2: Request-Reply Overview**

The request-reply pattern has two roles: The requester (service consumer or client) sends a request message and waits for a reply message. The replier (service provider) receives the request message and responds with a reply message.

Using the request-reply pattern with a Replier is straightforward. Connext DDS provides two Entities: the Requester and the Replier manage all the interactions on behalf of the application. The Requester and Replier automatically discover each other based on an application-specified service name. When the application invokes a request, the Requester sends a message (on an automatically-created request Topic) to the Replier, which notifies the receiving application. The application, in turn, uses the Replier to receive the
request and send the reply message. The reply message is sent by Connext DDS back to the original Requester (using a different automatically created reply Topic).

Connext DDS supports both blocking and non-blocking request-reply interactions:

- In a blocking (a.k.a. synchronous) interaction, the requesting application blocks while waiting for the reply. This is typical of applications desiring remote-procedure-call or remote-method-invocation interactions.
- In a non-blocking (a.k.a. asynchronous) interaction, the requesting application can proceed with other work and gets notified when a reply is available.

27.2 Repliers on page 1015 explains how an application can use the methods provided by the Requester and the Replier to perform both blocking and non-blocking request-reply interactions.

The implementation of request-reply in Connext DDS is highly scalable. A Replier can receive requests from thousands of Requesters at the same time. Connext DDS will efficiently deliver each reply only to the original Requester, allowing the number of Requesters to grow without significantly impacting each other.

### 26.1.1 Request-Reply Correlation

An application might have multiple outstanding requests, all originating from the same Requester. This can be as a result of using a non-blocking request-reply interaction, or as a result of having multiple application threads using the same Requester. Because of this, Connext DDS provides a way for the application to correlate a reply with the request it is associated with. This meta-data is provided as part of a SampleInfo structure that accompanies the reply.

When using a blocking request operation, Connext DDS provides an easy-to-use API that automatically does the correlation for you.

### 26.2 Single-Request, Multiple-Replies

Connext DDS also supports the single-request multiple-reply pattern. This pattern is an extension of the basic request-reply pattern in which multiple reply messages can flow back as a result of a single request.

The single-request multiple-reply pattern is very useful when getting large amounts of data as a reply, such as when querying a system for all data that matches a certain criteria. Another common use-case is invoking a service that goes through multiple stages and provides updates on each: service commencement, progress reports, and final completion.
For example, a mobile asset management system may need to locate a particular asset (truck, locomotive, etc.). The system sends out the request. The first reply that comes back will read “locating.” The service has not yet determined the position, but it notifies the requester that the search operation has started. The second reply might provide a status update on the search, perhaps including a rough area of location. The third and final reply will have the exact location of the asset.

### 26.3 Multiple Repliers

*Connext DDS* directly supports applications that obtain results from multiple providers in parallel instead of in sequence, basically implementing functional parallelism.

To illustrate, consider a system managing a fleet of drones, like unmanned aerial vehicles (UAVs). Using the single request-multiple reply pattern, the application can use a *Requester* to send a single ‘DroneInfo’ request to all the drones to query for their current mission and status. Each drone replies with the information on its own status and the *Requester* aggregates all the responses for the application.

As another example, consider a system that would like to locate the best printer to perform a particular job. The application can use a *Requester* to query all the printers that are on-line for their characteristics and load. The *Requester* receives the replies and accumulates them until an application-specified number of replies is received (or a timeout elapses). The application can then use the *Requester* to access all the replies, examine their contents, and select the best printer for the job.
26.4 Combining Request-Reply and Publish-Subscribe

Under the hood, Connext DDS implements request-reply using the DDS data-centric publish-subscribe pattern. This has a key benefit in that the two patterns can be combined, and mapped without interference.
For example, a pair of applications may be involved in a two-way conversation using request-reply. For debugging purposes or regulatory compliance, you want to inspect those request-reply messages, but without disrupting the conversation.

Since Connext DDS implements requests and replies using DDS data-centric publish-subscribe, others can simply subscribe to the request and reply messages. You can introduce a subscriber to the reply Topic, without interfering with the two-way conversation between the Requester and the Replier. This pattern is also known as a Wire Tap. For example, you can use RTI Recording Service to non-intrusively capture request-reply traffic.
Chapter 27 Using the Request-Reply Communication Pattern

There are two basic Connext DDS entities used by the Request-Reply communication pattern: Requester and Replier.

- A Requester publishes a request Topic and subscribes to a reply Topic. See 27.1 Requesters on the next page.
- A Replier subscribes to the request Topic and publishes the reply Topic. See 27.2 Repliers on page 1015.

There is an alternate type of replier known as a SimpleReplier:

- A SimpleReplier is useful for cases where there is a single reply to each request and the reply can be generated quickly, such as looking up some data from memory.
- A SimpleReplier is used in combination with a user-provided SimpleReplierListener. Requests are passed to a callback in the SimpleReplierListener, which returns the reply.
- The SimpleReplier is not suitable if the replier needs to generate more than one reply for a single request or if generating the reply can take significant time or needs to occur asynchronously. For more information, see 27.3 SimpleRepliers on page 1021.

Additional resources. In addition to the information in this section, you can find more information and example code here:

- The Connext DDS API Reference HTML documentation\(^1\) contains example code that will show you how to use the API: From the Modules tab, navigate to Programming How-

\(^1\)The API Reference HTML documentation is available for all supported programming languages. Open <NDDSHOME>/README.html.
To’s, Request-Reply Examples.

- The Connext DDS API Reference HTML documentation also contains the full API documentation for the Requester, Replier, and SimpleReplier. Under the Modules tab, navigate to RTI Connext DDS API Reference, RTI Connext Messaging API Reference, Request-Reply Pattern.

27.1 Requesters

A Requester is an entity with two associated DDS Entities: a DDS DataWriter bound to a request Topic and a DDS DataReader bound to a reply Topic. A Requester sends requests by publishing samples of the request Topic, and receives replies for those requests by subscribing to the reply Topic.

Valid types for request and reply Topics can be:

- For the C API:
  - DDS types generated by RTI Code Generator
- For all other APIs:
  - DDS types generated by RTI Code Generator
  - Built-in DDS types, such as, String, KeyedString, Octets, and KeyedOctets
  - DDS DynamicData Types

To communicate, a Requester and Replier must use the same request Topic name, the same reply Topic name, and be associated with the same DDS domain_id.

A Requester has an associated DomainParticipant, which can be shared with other requesters or Connext DDS entities. All the other entities required for request-reply interaction, including the request and reply Topics, the DataWriter for writing requests, and a DataReader for reading replies, are automatically created when the Requester is constructed.

Connext DDS guarantees that a Requester will only receive replies associated with the requests it sends.

The Requester uses the underlying DataReader not only to receive the replies, but also as a cache that can hold replies to multiple outstanding requests or even multiple replies to a single request. Depending on the HistoryQoSPolicy configuration of the DataReader, the Requester may allow replies to replace previous replies based on the reply data having the same value for the Key fields (see 2.4 DDS Samples, Instances, and Keys on page 18). The default configuration of the Requester does not allow replacing.

You can configure the QoS for the underlying DataWriter and DataReader in a QoS profile. By default, the DataWriter and DataReader are created with default values (DDS_DATAWRITER_QOS_DEFAULT and DDS_DATAREADER_QOS_DEFAULT, respectively) except for the following:
7.5.21 RELIABILITY QosPolicy on page 447: **kind** is set to RELIABLE.

7.5.12 HISTORY QosPolicy on page 420: **kind** is set to KEEP_ALL.

Several other protocol-related settings for Requesters (see the API Reference HTML documentation: select ** Modules, Programming How-To's, Request-Reply Examples**; then scroll down to the section on **Configuring Request-Reply QoS profiles**).

### 27.1.1 Creating a Requester

Before you can create a Requester, you need a DomainParticipant and a service name.

**Note:** The example code snippets in this section use the C++ API. You can find more complete examples in all the supported programming languages (C, C++, Java, C#) in the *Connext DDS* API Reference HTML documentation and in the “example” directory found in your *Connext DDS* installation.

To create a Requester with the minimum set of parameters, you can use the basic constructor that receives only an existing DDS DomainParticipant and the name of the service:

```
Requester<
MyRequestType, MyReplyType>
*requester =
new Requester<
MyRequestType, MyReplyType>(
participant, "ServiceName");
```

To create a Requester with specific parameters, you may use a different constructor that receives a RequesterParams structure (described in **27.1.3 Setting Requester Parameters on the next page**):

```
Requester( const RequesterParams &params)
```

The **ServiceName** parameter is used to generate the names of the request and reply Topics that the Requester and Replier will use to communicate. For example, if the service name is “MyService”, the topic names for the Requester and Replier will be “MyServiceRequest” and “MyServiceReply”, respectively. Therefore, for communication to occur, you must use the same service name when creating the Requester and the Replier entities.

If you want to use topic names different from the ones that would be derived from the ServiceName, you can override the default names by setting the actual request and reply Topic names using the **request_topic_name()** and **reply_topic_name()** accessors to the RequesterParams structure prior to creating the Requester.

**Example:** To create a Requester with default QoS and topic names derived from the service name, you may use the following code:

```
Requester<Foo, Bar> * requeste =
new Requester<Foo, Bar>(
participant, "MyService");
```

**Example:** To create a Requester with a specific QoS profile with library name “MyLibrary” and profile “MyProfile” defined inside USER_QOS_PROFILES.xml in the current working directory, you may use the following code:
Destroying a Requester

To destroy a Requester and free its underlying entities you may use the destructor:

```cpp
virtual ~Requester ()
```

## 27.1.3 Setting Requester Parameters

To change the RequesterParams that can be used when creating a Requester, you can use the operations listed in Table 27.1 Operations to Set Requester Parameters.

### Table 27.1 Operations to Set Requester Parameters

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>datareader_qos</code></td>
<td>Sets the QoS of the reply DataReader.</td>
</tr>
<tr>
<td><code>datawriter_qos</code></td>
<td>Sets the QoS of the request DataWriter.</td>
</tr>
<tr>
<td><code>publisher</code></td>
<td>Sets a specific Publisher.</td>
</tr>
<tr>
<td><code>qos_profile</code></td>
<td>Sets a QoS profile for the DDS entities in this requester.</td>
</tr>
<tr>
<td><code>request_topic_name</code></td>
<td>Sets the name of the Topic used for the request. If this parameter is set,</td>
</tr>
<tr>
<td></td>
<td>then you must also set the reply_topic_name parameter and you should not</td>
</tr>
<tr>
<td></td>
<td>set the service_name parameter.</td>
</tr>
<tr>
<td><code>reply_topic_name</code></td>
<td>Sets the name of the Topic used for the reply. If this parameter is set,</td>
</tr>
<tr>
<td></td>
<td>then you must also set the request_topic_name parameter and you should</td>
</tr>
<tr>
<td></td>
<td>not set the service_name parameter.</td>
</tr>
<tr>
<td><code>reply_type_support</code></td>
<td>Sets the type support for the reply type.</td>
</tr>
<tr>
<td><code>request_type_support</code></td>
<td>Sets the type support for the request type.</td>
</tr>
<tr>
<td><code>service_name</code></td>
<td>Sets the service name. This will automatically set the name of the request</td>
</tr>
<tr>
<td></td>
<td>Topic and the reply Topic. If this parameter is set you should not set</td>
</tr>
<tr>
<td></td>
<td>the request_topic_name or the reply_topic_name.</td>
</tr>
<tr>
<td><code>subscriber</code></td>
<td>Sets a specific Subscriber.</td>
</tr>
</tbody>
</table>

## 27.1.4 Summary of Requester Operations

There are several kinds of operations an application can perform using the `Requester`: 
27.1.4 Summary of Requester Operations

- Sending requests (i.e., publishing request samples on the request Topic)
- Waiting for replies to be received.
- Taking the reply data. This gets the reply data from the Requester and removes from the Requester cache.
- Reading the reply data. This gets the reply data from the Requester but leaves it in the Requester cache so it remain accessible to future operations on the Requester.
- Receiving replies (a convenience operation that is a combination of ‘waiting’ and ‘taking’ the data in a single operation)

These operations are summarized in Table 27.2 Requester Operations

<table>
<thead>
<tr>
<th>Table 27.2 Requester Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation</strong></td>
</tr>
<tr>
<td>Sending Requests</td>
</tr>
<tr>
<td>Waiting for Replies</td>
</tr>
<tr>
<td>Taking Reply Data</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Reading Reply Data</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Receiving Replies</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>
### 27.1.5 Sending Requests

To send a request, use the `send_request()` operation on the `Requester`. There are three variants of this operation, depending on the parameters that are passed in:

1. `send_request (const TRequest &request)`
2. `send_request (WriteSample<TRequest> &request)`
3. `send_request (WriteSampleRef<TRequest> &request)`

The first variant simply sends a request.

The second variant sends a request and gets back information about the request in a `WriteSample` container. This information can be used to correlate the request with future replies.

The third variant is just like the second, but puts the information in a `WriteSampleRef`, which holds references to the data and parameters. Both `WriteSample` and `WriteSampleRef` provide information about the request that can be used to correlate the request with future replies.

### 27.1.6 Processing Incoming Replies with a Requester

The `Requester` provides several operations that can be used to wait for and access replies:

- `wait_for_replies()`, see 27.1.6.1 Waiting for Replies on the next page
- `take_reply()`, `take_replies()`, `read_reply()` and `read_replies()`, see 27.1.6.2 Getting Replies on the next page
- `receive_reply()` and `receive_replies()`, see 27.1.6.3 Receiving Replies on page 1014

The `wait_for_replies` operations are used to wait until the replies arrive.

The `take_reply`, `take_replies`, `read_reply`, and `read_replies()` operations access the replies once they have arrived.
27.1.6 Processing Incoming Replies with a Requester

The `receive_reply` and `receive_replies` are convenience functions that combine waiting and accessing the replies and are equivalent to calling the ‘wait’ operation followed by the corresponding `take_reply` or `take_replies` operations.

Each of these operations has several variants, depending on the parameters that are passed in.

### 27.1.6.1 Waiting for Replies

Use the `wait_for_replies()` operation on the `Requester` to wait for the replies to previously sent requests. There are three variants of this operation, depending on the parameters that are passed in. All these variants block the calling thread until either there are replies or a timeout occurs.

1. `wait_for_replies (const DDS_Duration_t &max_wait)`
2. `wait_for_replies (int min_count,`
   `   const DDS_Duration_t &max_wait)`
3. `wait_for_replies (int min_count,`
   `   const DDS_Duration_t &max_wait,`
   `   const SampleIdentity_t &related_request_id)`

The first variant (only passing in `max_wait`) blocks until a reply is available or until `max_wait` time has elapsed, whichever comes first. The reply can be to any of the requests made by the `Requester`.

The second variant (passing in `min_count` and `max_wait`) blocks until at least `min_count` replies are available or until `max_wait` time has elapsed, whichever comes first. These replies may all be to the same request or to different requests made by the `Requester`.

The third variant (passing in `min_count`, `max_wait`, and `related_request_id`) blocks until at least `min_count` replies to the request identified by the `related_request_id` are available, or until `max_wait` time has passed, whichever comes first. Note that unlike the previous variants, the replies must all be to the same single request (identified by the `related_request_id`) made by the `Requester`.

Typically after waiting for replies, you will call `take_reply`, `take_replies`, `read_reply`, or `read_replies()`, see 27.2 Repliers on page 1015.

If you call `wait_for_replies()` several times without ‘taking’ the replies (using the `take_reply` or `take_replies` operation), future calls to `wait_for_replies()` will return immediately and will not wait for new replies.

### 27.1.6.2 Getting Replies

You can use the following operations to access replies: `take_reply`, `take_replies`, `read_reply`, and `read_replies()`.

As mentioned in 27.1.4 Summary of Requester Operations on page 1009, the difference between the ‘take’ operations (`take_reply`, `take_replies`) and the ‘read’ operations (`read_reply`, `read_replies`) is that ‘take’ operations remove the replies from the `Requester` cache. This means that future calls to `take_reply`, `read_reply`, `read_reply`, and `read_reply` will not get the same reply again.
The **take_reply** and **read_reply** operations access a *single* reply, whereas the **take_replies** and **read_replies** can access a *collection* of replies.

There are four variants of the **take_reply** and **read_reply** operations, depending on the parameters that are passed in:

<table>
<thead>
<tr>
<th>Variant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. take_reply (Sample&lt;TReply&gt; &amp;reply) &lt;br&gt; read_reply (Sample&lt;TReply&gt; &amp;reply)</td>
<td></td>
</tr>
<tr>
<td>2. take_reply (SampleRef&lt;TReply&gt; reply) &lt;br&gt; read_reply (SampleRef&lt;TReply&gt; reply)</td>
<td></td>
</tr>
<tr>
<td>3. take_reply (Sample&lt;TReply&gt; &amp;reply, &lt;br&gt; const SampleIdentity_t &amp;related_request_id) &lt;br&gt; read_reply (Sample&lt;TReply&gt; &amp;reply, &lt;br&gt; const SampleIdentity_t &amp;related_request_id)</td>
<td></td>
</tr>
<tr>
<td>4. take_reply (SampleRef&lt;TReply&gt; reply, &lt;br&gt; const SampleIdentity_t &amp;related_request_id) &lt;br&gt; read_reply (SampleRef&lt;TReply&gt; reply, &lt;br&gt; const SampleIdentity_t &amp;related_request_id)</td>
<td></td>
</tr>
</tbody>
</table>

The first two variants provide access to the next reply in the *Requester* cache. This is the earliest reply to any previous requests sent by the *Requester* that has not been ‘taken’ from the *Requester* cache. The remaining two variants provide access to the earliest non-previous ‘taken’ reply to the request specified by the **related_request_id**.

Notice that some of these variants use a *Sample*, while other use a *SampleRef*. A *SampleRef* can be used much like a *Sample*, but it holds references to the reply data and DDS *SampleInfo*, so there is no additional copy. In contrast using the *Sample* obtains a copy of both the data and DDS *SampleInfo*.

The **take_replies** and **read_replies** operations access a collection of (one or more) replies to previously sent requests. These operations are convenient when you expect multiple replies to a single request, or when issuing multiple requests concurrently without waiting for intervening replies.

The **take_replies** and **read_replies** operations return a *LoanedSamples* container that holds the replies. To increase performance, the *LoanedSamples* does not copy the reply data. Instead it ‘loans’ the necessary resources from the *Requester*. The resources loaned by the *LoanedSamples* container must be eventually returned, either explicitly calling the **return_loan()** operation on the *LoanedSamples* or through the destructor of the *LoanedSamples*.

There are three variants of the **take_replies** and **read_replies** operations, depending on the parameters that are passed in:

<table>
<thead>
<tr>
<th>Variant</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. take_replies (int max_count=DDS_LENGTH_UNLIMITED) &lt;br&gt; read_replies (int max_count=DDS_LENGTH_UNLIMITED)</td>
<td></td>
</tr>
<tr>
<td>2. take_replies (int max_count, &lt;br&gt; const SampleIdentity_t &amp;related_request_id) &lt;br&gt; read_replies (int max_count, &lt;br&gt; const SampleIdentity_t &amp;related_request_id)</td>
<td></td>
</tr>
</tbody>
</table>
3. take_replies (const SampleIdentity_t &related_request_id)
   read_replies (const SampleIdentity_t &related_request_id)

The first variant (only passing in max_count) returns a container holding up to max_count replies.

The second variant (passing in max_count and related_request_id) returns a LoanedSamples container holding up to max_count replies that correspond to the request identified by the related_request_id.

The third variant (only passing in related_request_id) returns a LoanedSamples container holding an unbounded number of replies that correspond to the request identified by the related_request_id. This is equivalent to the second variant with max_count = DDS_LENGTH_UNLIMITED.

The resources for the LoanedSamples container must be eventually be returned, either by calling the return_loan() operation on the LoanedSamples or through the LoanedSamples destructor.

For multi-reply scenarios, in which a Requester receives multiple replies from a Replier for a given request, the Requester can check if a reply is the last reply in a sequence of replies. To do so, see if the bit INTERMEDIATE_REPLY_SEQUENCE_SAMPLE is set in DDS_SampleInfo’s flag field (see Table 8.17 DDS_SampleInfo Structure) after receiving each reply. This bit indicates it is NOT the last reply.

### 27.1.6.3 Receiving Replies

The receive_reply() operation is a shortcut that combines calls to wait_for_replies() and to take_reply(). Similarly the receive_replies() operation combines wait_for_replies() and take_replies().

There is only one variant of the receive_reply() operation:

1. receive_reply (Sample<TReply> &reply, const DDS_Duration_t &timeout)

This operation blocks until either a reply is received or a timeout occurs. The contents of the reply are copied into the provided sample (reply).

There are two variants of the receive_replies() operation, depending on the parameters that are passed in:

1. receive_replies (const DDS_Duration_t &max_wait)

2. receive_replies (int min_count, int max_count,
               const DDS_Duration_t &max_wait)

These two variants block until multiple replies are available or a timeout occurs.

The first variant (only passing in max_wait) blocks until at least one reply is available or until max_wait time has passed, whichever comes first. The operation returns a LoanedSamples container holding the replies. Note that there could be more than one reply. This can occur if, for example, there were already replies available in the Requester from previous requests that were not processed. This operation does not limit the number of replies that can be returned on the LoanedSamples container.

The second variant (passing in min_count, max_count, and max_wait) will block until min_count replies are available or until max_wait time has passed, whichever comes first. Up to max_count replies will be stored into the LoanedSamples container which is returned to the caller.
The resources held in the LoanedSamples container must eventually be returned, either with an explicit call to return_loan() on the LoanedSamples or through the LoanedSamples destructor.

### 27.2 Repliers

A Replier is an entity with two associated DDS Entities: a DDS DataReader bound to a request Topic and a DDS DataWriter bound to a reply Topic. The Replier receives requests by subscribing to the request Topic and sends replies to those requests by publishing on the reply Topic.

Valid data types for these topics are the same as specified for the Requester, see 27.1 Requesters on page 1007.

For multi-reply scenarios in which a Replier generates more than one reply for a request, the Replier should mark all intermediate replies (all but the last reply) with the INTERMEDIATE_REPLY_SEQUENCE_SAMPLE bit-flag in the WriteParams_t flag field (see Table 7.17 DDS_WriteParams_t).

Much like a Requester, a Replier has an associated DDS DomainParticipant which can be shared with other Connext DDS entities. All the other entities required for the request-reply interaction, including a DataWriter for writing replies and a DataReader for reading requests, are automatically created when the Replier is constructed.

You can configure the QoS for the underlying DataWriter and DataReader in a QoS profile. By default, the DataWriter and DataReader are created with default QoS values (using DDS_DATAWRITER_QOS_DEFAULT and DDS_DATAREADER_QOS_DEFAULT, respectively) except for the following:

- 7.5.21 RELIABILITY QosPolicy on page 447: kind is set to RELIABLE
- 7.5.12 HISTORY QosPolicy on page 420: kind is set to KEEP_ALL

The Replier API supports several ways in which the application can be notified of, and process, requests:

- **Blocking**: The application thread blocks waiting for requests, processes them, and dispatches the reply. In this situation, if the computation necessary to process the request and produce the reply is small, you may consider using the SimpleReplier, which offers a simplified API.
- **Polling**: The application thread checks (polls) for requests periodically but does not block to wait for them. To check for data without blocking, call take_requests() or read_requests().
- **Asynchronous notification**: The application installs a ReplierListener to receive notifications whenever a request is received.

#### 27.2.1 Creating a Replier

To create a Replier with the minimum set of parameters you can use the basic constructor that receives only an existing DDS DomainParticipant and the name of the service:
27.2.2 Destroying a Replier

Replier (DDSDomainParticipant * participant,
    const std::string & service_name)

Example:

Replier<Foo, Bar> * replier =
    new Replier<Foo, Bar>(participant, "MyService");

To create a Replier with specific parameters you may use a different constructor that receives a ReplierParams structure:

Replier (const ReplierParams<TRequest, TReply> &params)

Example:

Replier<Foo, Bar> * replier = new Replier<Foo, Bar>(
    ReplierParams(participant).service_name("MyService")
    .qos_profile("MyLibrary", "MyProfile"));

The service_name is used to generate the names of the request and reply Topics that the Requester and Replier will use to communicate. For example, if the service name is “MyService”, the topic names for the Requester and Replier will be “MyServiceRequest” and “MyServiceReply”, respectively. Therefore it is important to use the same service_name when creating the Requester and the Replier.

If you need to specify different Topic names, you can override the default names by setting the actual request and reply Topic names using request_topic_name() and reply_topic_name() accessors to the ReplierParams structure prior to creating the Replier.

27.2.2 Destroying a Replier

To destroy a Replier and free its underlying entities:

virtual ~Replier ()

27.2.3 Setting Replier Parameters

To change the ReplierParams that are used to create a Replier, use the operations listed in Table 27.3 Operations to Set Replier Parameters.

Table 27.3 Operations to Set Replier Parameters

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datareader_qos</td>
<td>Sets the quality of service of the request DataReader.</td>
</tr>
<tr>
<td>datawriter_qos</td>
<td>Sets the quality of service of the reply DataWriter.</td>
</tr>
<tr>
<td>publisher</td>
<td>Sets a specific Publisher.</td>
</tr>
<tr>
<td>qos_profile</td>
<td>Sets a QoS profile for the entities in this replier.</td>
</tr>
<tr>
<td>replier_listener</td>
<td>Sets a listener that is called when requests are available.</td>
</tr>
</tbody>
</table>
Table 27.3 Operations to Set Replier Parameters

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>reply_topic_name</td>
<td>Sets a specific reply topic name.</td>
</tr>
<tr>
<td>reply_type_support</td>
<td>Sets the type support for the reply type.</td>
</tr>
<tr>
<td>request_topic_name</td>
<td>Sets a specific request topic name.</td>
</tr>
<tr>
<td>request_type_support</td>
<td>Sets the type support for the request type.</td>
</tr>
<tr>
<td>service_name</td>
<td>Sets the service name the Replier offers and Requesters use to match.</td>
</tr>
<tr>
<td>subscriber</td>
<td>Sets a specific Subscriber.</td>
</tr>
</tbody>
</table>

27.2.4 Summary of Replier Operations

There are four kinds of operations an application can perform using the Replier:

- Waiting for requests to be received
- Reading/taking the request data and associated information
- Receiving requests (a convenience operation that combines waiting and getting the data into a single operation)
- Sending a reply for received request (i.e., publishing a reply sample on the reply Topic with special meta-data so that the original Requester can identify it).

The Replier operations are summarized in Table 27.4 Replier Operations.

Table 27.4 Replier Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting for Requests</td>
<td>wait_for_requests</td>
<td>Waits for requests.</td>
</tr>
<tr>
<td>Taking Requests</td>
<td>take_request</td>
<td>Copies the contents of a single request into a Sample and removes it from the Replier cache.</td>
</tr>
<tr>
<td></td>
<td>take_requests</td>
<td>Returns a LoanedSample to access multiple requests and removes the requests from the Replier cache.</td>
</tr>
<tr>
<td>Reading Requests</td>
<td>read_request</td>
<td>Copies the contents of a single request into a Sample, leaving it in the Replier cache</td>
</tr>
<tr>
<td></td>
<td>read_requests</td>
<td>Returns a LoanedSample to access multiple requests, leaving them in the Replier cache.</td>
</tr>
</tbody>
</table>
### Table 27.4 Replier Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Receiving Requests</td>
<td>receive_request</td>
<td>Waits for a single request and copies its contents into a Sample container.</td>
</tr>
<tr>
<td></td>
<td>receive_requests</td>
<td>Waits for multiple requests and provides a LoanedSamples container to access them.</td>
</tr>
<tr>
<td>Sending Replies</td>
<td>send_reply</td>
<td>Sends a reply for a previous request.</td>
</tr>
<tr>
<td>Getting Underlying Entities</td>
<td>get_request_datareader</td>
<td>Retrieves the underlying DataReader.</td>
</tr>
<tr>
<td></td>
<td>get_reply_datawriter</td>
<td>Retrieves the underlying DataWriter.</td>
</tr>
</tbody>
</table>

#### 27.2.5 Processing Incoming Requests with a Replier

The Replier provides several operations that can be used to wait for and access the requests:

- **wait_for_requests()**, see 27.2.5.1 Waiting for Requests below
- **take_request(), take_requests(), read_request(), and read_requests()**, see 27.2.5.2 Reading and Taking Requests on the next page
- **receive_request()** and **receive_requests()**, see 27.2.5.3 Receiving Requests on page 1020

The **wait_for_requests()** operations are used to wait until requests arrive.

The **take_request(), take_requests(), read_request(), and read_requests()** operations access the requests, once they have arrived.

The **receive_request()** and **receive_requests()** operations are convenience functions that combine waiting for and accessing requests and are equivalent to calling the ‘wait’ operation followed by the corresponding **take_request()** or **take_requests()** operations.

Each of these operations has several variants, depending on the parameters that are passed in.

#### 27.2.5.1 Waiting for Requests

Use the **wait_for_requests()** operation on the Replier to wait for requests. There are two variants of this operation, depending on the parameters that are passed in. All these variants block the calling thread until either there are replies or a timeout occurs:

1. `wait_for_requests (const DDS_Duration_t &max_wait)`
2. `wait_for_requests (int min_count, const DDS_Duration_t &max_wait)`
27.2.5 Processing Incoming Requests with a Replier

The first variant (only passing in max_wait) blocks until one request is available or until max_wait time has passed, whichever comes first.

The second variant blocks until min_count number of requests are available or until max_wait time has passed.

Typically after waiting for requests, you will call take_request, take_requests, read_request, or read_requests, see 27.2.6 Sending Replies on the next page.

27.2.5.2 Reading and Taking Requests

You can use the following four operations to access requests: take_request, take_requests, read_request, or read_requests.

As mentioned in 27.2.4 Summary of Replier Operations on page 1017, the difference between the ‘take’ operations (take_request, take_requests) and the ‘read’ operations (read_request, read_requests) is that ‘take’ operations remove the requests from the Replier cache. This means that future calls to take_request, take_requests, read_request, or read_requests will not get the same request again.

The take_request and read_request operations access a single reply, whereas the take_requests and read_requests can access a collection of replies.

There are two variants of the take_request and read_request operations, depending on the parameters that are passed in:

1. take_request (connext::Sample<TRequest> & request)
   read_request (connext::Sample<TRequest> & request)

2. take_request (connext::SampleRef<TRequest> request)
   read_request (connext::SampleRef<TRequest> request)

The first variant returns the request using a Sample container. The second variant uses a SampleRef container instead. A SampleRef can be used much like a Sample, but it holds references to the request data and DDS SampleInfo, so there is no additional copy. In contrast, using the Sample makes a copy of both the data and DDS SampleInfo.

The take_requests and read_requests operations access a collection of (one or more) requests in the Replier cache. These operations are convenient when you want to batch-process a set of requests.

The take_requests and read_requests operations return a LoanedSamples container that holds the requests. To increase performance, the LoanedSamples does not copy the request data. Instead it ‘loans’ the necessary resources from the Replier. The resources loaned by the LoanedSamples container must be eventually returned, either explicitly by calling the return_loan() operation on the LoanedSamples or through the destructor of the LoanedSamples.

There is only one variant of these operations:

1. take_requests (int max_samples = DDS_LENGTH_UNLIMITED)
   read_requests (int max_samples = DDS_LENGTH_UNLIMITED)
The returned container may contain up to \texttt{max_samples} number of requests.

### 27.2.5.3 Receiving Requests

The \texttt{receive\_request()} operation is a shortcut that combines calls to \texttt{wait\_for\_requests()} and \texttt{take\_request()}. Similarly, the \texttt{receive\_requests()} operation combines \texttt{wait\_for\_requests()} and \texttt{take\_requests()}.

There are two variants of the \texttt{receive\_request()} operation:

1. \texttt{receive\_request (connect::Sample<TRequest> \& request, const DDS\_Duration\_t \& max\_wait)}
2. \texttt{receive\_request (connect::SampleRef<TRequest> request, const DDS\_Duration\_t \& max\_wait)}

The \texttt{receive\_request} operation blocks until either a request is received or a timeout occurs. The contents of the request are copied into the provided container (\texttt{request}). The first variant uses a \texttt{Sample} container, whereas the second variant uses a \texttt{SampleRef} container. A \texttt{SampleRef} can be used much like a \texttt{Sample}, but it holds \texttt{references} to the request data and \texttt{DDS SampleInfo}, so there is no additional copy. In contrast, using the \texttt{Sample} obtains a copy of both the data and the \texttt{DDS SampleInfo}.

There are two variants of the \texttt{receive\_requests()} operation, depending on the parameters that are passed in:

1. \texttt{receive\_requests (const DDS\_Duration\_t \& max\_wait)}
2. \texttt{receive\_requests (int min\_request\_count, int max\_request\_count, const DDS\_Duration\_t \& max\_wait)}

The \texttt{receive\_requests} operation blocks until one or more requests are available, or a timeout occurs.

The first variant (only passing in \texttt{max\_wait}) blocks until one request is available or until \texttt{max\_wait} time has passed, whichever comes first. The contents of the request are copied into a \texttt{LoanedSamples} container which is returned to the caller. An unlimited number of replies can be copied into the container.

The second variant blocks until \texttt{min\_request\_count} number of requests are available or until \texttt{max\_wait} time has passed, whichever comes first. Up to \texttt{max\_request\_count} number of requests will be copied into a \texttt{LoanedSamples} container which is returned to the caller.

The resources for the \texttt{LoanedSamples} container must eventually be returned, either with \texttt{return\_loan()} or through the \texttt{LoanedSamples} destructor.

### 27.2.6 Sending Replies

There are three variants for \texttt{send\_reply()}, depending on the parameters that are passed in:
1. `send_reply` (const `TReply` & reply, 
   const `SampleIdentity_t` & related_request_id)
2. `send_reply` (WriteSample<TReply> & reply, 
   const `SampleIdentity_t` & related_request_id)
3. `send_reply` (WriteSampleRef<TReply> & reply, 
   const `SampleIdentity_t` & related_request_id)

This operation sends a reply for a previous request. The related request ID can be retrieved from an existing request Sample.

The first variant is recommended if you do not need to change any of the default write parameters.

The other two variants allow you to set custom parameters for writing a reply. Unlike the Requester, where retrieving the sample ID for correlation is common, on the Replier side using a WriteSample or WriteSampleRef is only necessary when you need to overwrite the default write parameters. If that’s not the case, use the first variant.

One reason to override the default write parameters is a multi-reply scenario in which a Replier generates more than one reply for a request. In this case, all the intermediate replies (all but the last reply) should be marked with the INTERMEDIATE_REPLY_SEQUENCE_SAMPLE bit-flag in the flag field within WriteSample::info or WriteSampleRef::info.

A Requester can detect if a reply is the last reply in a sequence of replies by seeing if INTERMEDIATE_REPLY_SEQUENCE_SAMPLE is NOT set in the flag field of Sample::info after receiving each reply.

### 27.3 SimpleRepliers

The SimpleReplier offers a simplified API to receive and process requests. The API is based on a user-provided object that implements the SimpleReplierListener interface. Requests are passed to the listener operation implemented by the user-provided object, which processes the request and returns a reply.

The SimpleReplier is recommended if each request generates a single reply and computing the reply can be done quickly with very little CPU resources and without calling any operations that may block the processing thread. For example, looking something up in an internal memory-based data structure would be a good use case for using a SimpleReplier.

#### 27.3.1 Creating a SimpleReplier

To create a SimpleReplier with the minimum set of parameters, you can use the basic constructor:

```c++
SimpleReplier (DDSDomainParticipant *participant, 
   const std::string &service_name, 
   SimpleReplierListener<TRequest, TReply> &listener)
```

To create a SimpleReplier with specific parameters, you may use a different constructor that receives a SimpleReplierParams structure:
27.3.2 Destroying a SimpleReplier

To destroy a SimpleReplier and free its resources use the destructor:

```cpp
virtual ~SimpleReplier ()
```

27.3.3 Setting SimpleReplier Parameters

To change the SimpleReplierParams used to create a SimpleReplier, use the operations in Table 27.5 Operations to Set SimpleReplier Parameters.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>datareader_qos</td>
<td>Sets the quality of service of the reply DataReader.</td>
</tr>
<tr>
<td>datawriter_qos</td>
<td>Sets the quality of service of the reply DataWriter.</td>
</tr>
<tr>
<td>publisher</td>
<td>Sets a specific Publisher.</td>
</tr>
<tr>
<td>qos_profile</td>
<td>Sets a QoS profile for the entities in this replier.</td>
</tr>
<tr>
<td>reply_topic_name</td>
<td>Sets a specific reply topic name.</td>
</tr>
<tr>
<td>reply_type_support</td>
<td>Sets the type support for the reply type.</td>
</tr>
<tr>
<td>request_topic_name</td>
<td>Sets a specific request topic name.</td>
</tr>
<tr>
<td>request_type_support</td>
<td>Sets the type support for the request type.</td>
</tr>
<tr>
<td>service_name</td>
<td>Sets the service name the Replier offers and Requesters use to match.</td>
</tr>
<tr>
<td>subscriber</td>
<td>Sets a specific Subscriber.</td>
</tr>
</tbody>
</table>

27.3.4 Getting Requests and Sending Replies with a SimpleReplierListener

The on_request_available() operation on the SimpleReplierListener receives a request and returns a reply.

```cpp
on_request_available(TRequest &request)
```

This operation gets called when a request is available. It should immediately return a reply. After calling on_request_available(), Connext DDS will call the operation return_loan() on the SimpleReplierListener; this gives the application-defined listener an opportunity to release any resources related to computing the previous reply.
27.4 Accessing Underlying DataWriters and DataReaders

Both Requester and Replier entities have underlying DDS DataWriter and DataReader entities. These are created automatically when the Requester and Replier are constructed.

Accessing the DataWriter used by a Requester may be useful for a number of advanced use cases, such as:

- Finding matching subscriptions (e.g., Replier entities), see 7.3.16.1 Finding Matching Subscriptions on page 342
- Setting a DataWriterListener, see 7.3.4 Setting Up DataWriterListeners on page 294
- Getting DataWriter protocol or cache statuses, see 7.3.6 Statuses for DataWriters on page 296
- Flushing a data batch after sending a number of request samples, see 7.3.9 Flushing Batches of DDS Data Samples on page 315
- Modifying the QoS

Accessing the reply DataReader may be useful for a number of advanced use cases, such as:

- Finding matching publications (e.g., Requester entities), see 8.3.10 Navigating Relationships Among Entities on page 551
- Getting DataReader protocol or cache statuses, see 8.3.5 Checking DataReader Status and StatusConditions on page 514 and 8.3.7 Statuses for DataReaders on page 516.
- Modifying the QoS

To access these underlying objects:

- RequestDataWriter * get_request_datawriter()
- RequestDataReader * get_request_datareader()
- ReplyDataWriter * get_reply_datawriter()
- ReplyDataReader * get_reply_datareader()
Chapter 28 Remote Procedure Calls (RPC)—Experimental Feature

Remote Procedure Calls, or RPC, is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space.

*Note:* RPC is an experimental feature available only on C++11, for certain platforms. See the RTI Connext DDS Core Libraries Platform Notes for the supported architectures. See also Experimental Features in the RTI Connext DDS Core Libraries Release Notes.

RPC has two participants: a *client* and a *service*. Under the hood, the *client* uses a Requester to send requests and receive replies; the *service* uses a Replier to receive the requests and send the replies.

RPC over DDS uses a function-call style where the *client/Requester* directly calls the *service/Replier* by calling the service's functions, abstracting sending the request and receiving the corresponding reply on the client side.

*Figure 28.1: RPC Overview*
Connext DDS supports both blocking and non-blocking interactions:

- In a blocking (or synchronous) interaction, the client application blocks while waiting for the service’s answer.
- In a non-blocking (or asynchronous) interaction, the client application can proceed with other work, ask if the service’s answer is already available, or wait for the service’s answer.

It is possible for a client to call more than one function at a time, particularly when asynchronous invocations are used.

28.2 RPC Client on page 1029 explains how a client application can use the method provided by the interface to perform both blocking and non-blocking interactions with the service.

Additional resources. In addition to the information in this section, you can find more information and example code here:

- The Connext DDS API Reference HTML documentation contains example code that will show you how to use the API: From the Modules tab, navigate to Programming How-To’s, RPC Tutorial.
- The Connext DDS API Reference HTML documentation also contains the full API documentation for the client-side and server-side APIs. Under the Modules tab, navigate to RTI Connext DDS API Reference, RTI Connext Messaging API Reference, Remote Procedure Call.

28.1 RPC Service

A service/Replier receives requests from those clients/Requesters that are subscribed to it, and sends replies to those clients. To communicate, a service and a client must use the same service name, and be associated with the same DDS domain_id.

A service has an associated Replier. All the entities required by the Replier—including the DomainParticipant, the request and reply Topics, the DataWriter for writing replies, and a DataReader for reading the requests—are automatically created when the service is created.

You can configure the QoS policies for the underlying DataWriter and DataReader by setting them either in the DomainParticipant that the service is using or in a QoS Profile.

A service definition in IDL is represented as an interface with the annotation @DDSService or @service\. An interface may define as many operations and attributes as you like. Exceptions are also supported by an interface and can be thrown by operations and attributes.

---

1The @service annotation receives the type of service as an argument. Connext DDS accepts only “DDS” or “*”. If no parameter is passed to the annotation, “*” is assumed.
Attributes are defined by using 'attribute' before the type of the attribute. For example: attribute float speed. Other elements inside an interface are considered operations. See 28.2.3 Summary of Client Operations on page 1030.

```c
module robot {
    exception TooFast {};
    enum Command { START_COMMAND, STOP_COMMAND };
    struct Status {
        string msg;
    };

    @DDSService
    interface RobotControl {
        void command(Command com);
        float setSpeed(float speed) raises (TooFast);
        float getSpeed();
        void getStatus(out Status status);
    };
}; //module robot
```

### 28.1.1 Creating a Service

Before you can create a service/Replier, you need a DomainParticipant, a Server, a ServiceParams, and an instance of the service interface.

A Server defines the execution context for one or more services. A Server is created with ServerParams, which allow configuring a thread pool that executes the services.

```c
dds::rpc::ServerParams server_params;
server_params.extensions().thread_pool_size(4);
dds::rpc::Server server(server_params);
```

A service requires a DomainParticipant and an identifier, which are specified (among other optional configuration parameters) using ServiceParams:

```c
dds::domain::DomainParticipant client_participant(domain_id);
dds::rpc::ServiceParams params(participant);
params.service_name("Example RobotControl Service");
```

The service_name function is used to generate the names of the request and reply Topics that the internal Requester (client) and Replier (service) will use to communicate. For example, if the service name is “MyService”, the Topic names for the Requester and Replier will be “MyServiceRequest” and “MyServiceReply”, respectively. Therefore, for communication to occur, you must use the same service name when creating the Requester and the Replier entities. If you want to use Topic names different from the ones that would be derived from the ServiceName, you can override the default names by setting the actual request and reply Topic names using the request_topic_name() and reply_topic_name() accessors to the RequesterParams structure before creating the Requester.

Next, create an instance of the interface implementation:
auto service_impl = std::make_shared<RobotControlExample>();

The *service* implementation contains the definition of each operation defined inside the interface in IDL. The *service/Replier* will call these methods upon receiving the request from the *client/Requester*.

Finally, create a *service* for the interface implementation, attached to the server and using the parameters specified before:

    RobotControlService service(service_impl, server, params);

The *service* is ready to receive function calls as soon as it is created.

You can create additional *services* and attach them to the same Server.

Note that just like *DomainParticipant*, 'RobotControlService' and 'server' are reference types and behave like shared pointers. You need to keep one or more references to them to avoid their destruction. You can also use the `run()` operation on the Server to block the current thread, and explicitly call `close()` to unblock it and destroy the Server (see Table 28.2 Server Operations).

### 28.1.2 Setting the Server Parameters

To change the ServerParameters that can be used when creating a Server (for the *service/Replier*), you can use the operations listed in Table 28.1 Operations to Set Server Parameters.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>thread_pool_size</td>
<td>Configures the number of threads of a Server thread pool.</td>
</tr>
<tr>
<td>async_waitset_property</td>
<td>Allows fine-tuning the internal AsyncWaitSet used to process function calls.</td>
</tr>
</tbody>
</table>

*Note: These operations are extensions, they must be called via `this->extensions()`.*

### 28.1.3 Summary of Server Operations

There are two kinds of operations an application can perform using the Server:

- Running the service
- Closing the service

The Server operations are summarized in Table 28.2 Server Operations.
Table 28.2 Server Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run the Server</td>
<td>run</td>
<td>Holds the execution of the current thread.</td>
</tr>
<tr>
<td>Close the Server</td>
<td>close</td>
<td>Unblocks run() and forces the destruction of this entity. Note that calling close() is not necessary since Server is a reference type and is destroyed automatically when no longer referenced.</td>
</tr>
</tbody>
</table>

28.1.4 Run the Server

The Server is ready to run as soon as it is created and one or more services/Repliers are attached to it. It doesn’t require any specific call to start running. However, Connext DDS provides two optional run() operations that simply hold the execution of the current thread:

- run (const dds::core::Duration &maxWait)
- run()

The first operation holds the execution of the current thread for the specified amount of time. The second operation holds the execution of the current thread for an unlimited period of time.

To close the Server, the Server provides a close() operation, which unlocks run() and then releases all resources.

28.1.5 Setting the Service Parameters

To change the ServiceParams that can be used when creating a service/Replier, you can use the operations listed in Table 28.3 Operations to Set Service Parameters.

Table 28.3 Operations to Set Service Parameters

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>service_name</td>
<td>The service name the Replier offers and Requesters use to match.</td>
</tr>
<tr>
<td>request_topic_name</td>
<td>Sets a specific request Topic name.</td>
</tr>
<tr>
<td>reply_topic_name</td>
<td>Sets a specific reply Topic name.</td>
</tr>
<tr>
<td>datawriter_qos</td>
<td>Sets the Quality of Service of the reply DataWriter.</td>
</tr>
<tr>
<td>datareader_qos</td>
<td>Sets the Quality of Service of the reply DataReader.</td>
</tr>
<tr>
<td>publisher</td>
<td>Sets a specific Publisher.</td>
</tr>
<tr>
<td>subscriber</td>
<td>Sets a specific Subscriber.</td>
</tr>
</tbody>
</table>
### 28.2 RPC Client

A *client* allows making remote function calls to the *services* that it is subscribed to, and receives the results to those calls from those *services*. To communicate, a *client* and a *service* must use the same *service* name, and be associated with the same DDS *domain_id*.

A *client* has an associated *Requester*. All the entities required by the *Requester*—including the *DomainParticipant*, the request and reply *Topics*, the *DataWriter* for writing the requests, and a *DataReader* for reading replies—are automatically created when the *client* is created.

You can configure the QoS for the underlying *DataWriter* and *DataReader* by setting them either in the *DomainParticipant* that the client is using or in a QoS Profile.

A client is also defined as an interface in IDL with the annotation `@DDSService` or `@service`.

#### 28.2.1 Creating a Client

To create a *client/Requester*, you need a `ClientParams`, a `DomainParticipant`, and a *service* name:

```cpp
dds::domain::DomainParticipant client_participant(domain_id);
dds::rpc::ClientParams client_params(client_participant);
client_params.service_name("Example RobotControl Service");
```

The *service_name* function is used to generate the names of the request and reply *Topics* that the internal *Requester* and *Replier* will use to communicate. For example, if the service name is “MyService”, the *Topic* names for the *Requester* and *Replier* will be “MyServiceRequest” and “MyServiceReply”, respectively. Therefore, for communication to occur, you must use the same service name when creating the *Requester* and the *Replier* entities. If you want to use *Topic* names different from the ones that would be derived from the ServiceName, you can override the default names by setting the actual request and reply *Topic* names using the `request_topic_name()` and `reply_topic_name()` accessors to the RequesterParams structure prior to creating the *Requester*.

To create a client, use the constructor that receives the `ClientParams`:

```cpp
RobotControlClient client(client_params);
```

Once you have created a client, you can use it to perform the operations in Table 28.5 Client Operations.

#### 28.2.2 Setting the Client Parameters

To change the `ClientParams` that can be used when creating a *client/Requester*, you can use the operations listed in Table 28.4 Operations to Set Client Parameters.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_type</td>
<td>The request type, when DynamicData is used.</td>
</tr>
<tr>
<td>reply_type</td>
<td>The reply type, when DynamicData is used.</td>
</tr>
</tbody>
</table>
Table 28.4 Operations to Set Client Parameters

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>function_call_max_wait</td>
<td>Specifies the maximum wait time for all the remote calls.</td>
</tr>
<tr>
<td>service_name</td>
<td>The service name that Replier and Requesters use to match and communicate.</td>
</tr>
<tr>
<td>request_topic_name</td>
<td>Sets a specific request Topic name.</td>
</tr>
<tr>
<td>reply_topic_name</td>
<td>Sets a specific reply Topic name.</td>
</tr>
<tr>
<td>datawriter_qos</td>
<td>Sets the Quality of Service of the request DataWriter.</td>
</tr>
<tr>
<td>datareader_qos</td>
<td>Sets the Quality of Service of the request DataReader.</td>
</tr>
<tr>
<td>publisher</td>
<td>Sets a specific Publisher.</td>
</tr>
<tr>
<td>subscriber</td>
<td>Sets a specific Subscriber.</td>
</tr>
<tr>
<td>request_type</td>
<td>The request type, when DynamicData is used.</td>
</tr>
<tr>
<td>reply_type</td>
<td>The reply type, when DynamicData is used.</td>
</tr>
</tbody>
</table>

28.2.3 Summary of Client Operations

There are several kinds of operations an application can perform using the client/Requester:

- Waiting for service to be discovered
- Making remote function calls (synchronous or asynchronous)

The Client operations are summarized in Table 28.5 Client Operations.

The <operation_name> comes from the IDL file. In the example IDL file in 28.1 RPC Service on page 1025, the client will have an operation with the same name as void command(Command com). In the case of an attribute, such as attribute long test, the client in C++ will have two operations, named long get_operation_test(); and void set_operation_test(long test);. The <attribute_name> is defined by adding 'attribute' before its type: attribute <type> <name>, such as attribute long test.

Table 28.5 Client Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting for Service</td>
<td>wait_for_service</td>
<td>Waits for services.</td>
</tr>
</tbody>
</table>
### 28.2.3 Summary of Client Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sending Request</td>
<td><code>&lt;operation_name&gt;</code></td>
<td>Makes a synchronous remote function call.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;get/set&gt;_attribute_&lt;attribute_name&gt;</code></td>
<td>28.2.3.2 Making Remote Function Calls below</td>
</tr>
<tr>
<td></td>
<td><code>&lt;operation_name&gt;_async</code></td>
<td>Makes an asynchronous remote function call.</td>
</tr>
<tr>
<td></td>
<td><code>&lt;get/set&gt;_attribute_&lt;attribute_name&gt;_async</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Getting Underlying Entities</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>request_datawriter</td>
<td>Retrieves the underlying DataWriter that sends the request.</td>
<td>28.3 Accessing Underlying DataWriters and DataReaders on the next page</td>
</tr>
<tr>
<td>reply_datareader</td>
<td>Retrieves the underlying DataReader that receives the replies.</td>
<td></td>
</tr>
</tbody>
</table>

#### 28.2.3.1 Waiting for Services

The client/Requester provides two operations that can be used to wait for services:

- `wait_for_service (const dds::core::Duration &maxWait)`
- `wait_for_service ()`

The first operation blocks until one service is available or until `maxWait` time has passed, whichever comes first. The second operation blocks until one service is available for an unlimited period of time.

#### 28.2.3.2 Making Remote Function Calls

To send a request, use the operations and attributes defined in the interface:

- `<operation_name>(<args>)`
- `get_attribute_<attribute_name>(<args>)`
- `set_attribute_<attribute_name>(<args>)`

These functions block the client until the service sends a reply.

The client also provides asynchronous functions that don’t block the client:

- `<operation_name>_async(<args>)`
- `get_attribute_<attribute_name>_async(<args>)`
- `set_attribute_<attribute_name>_async(<args>)`

Asynchronous functions return a `std::future` that will contain the result when it’s received:
The call to `std::future::get()` provides the result if it’s already available or blocks until it is.

## 28.3 Accessing Underlying DataWriters and DataReaders

Both *client* and *service* entities have underlying DDS *DataWriter* and *DataReader* entities. These are created automatically when the *client* and *service* are constructed.

Accessing the *DataWriter* used by a *client* may be useful for a number of advanced use cases, such as:

- Finding matching subscriptions (i.e., *service* entities), see 7.3.16.1 Finding Matching Subscriptions on page 342.
- Setting a *DataWriterListener*, see 7.3.4 Setting Up DataWriterListeners on page 294.
- Getting *DataWriter* protocol or cache statuses, see 7.3.6 Statuses for DataWriters on page 296.
- Flushing a data batch after sending a number of request samples, see 7.3.9 Flushing Batches of DDS Data Samples on page 315.
- Modifying the QoS.

Accessing the service *DataReader* may be useful for a number of advanced use cases, such as:

- Finding matching publications (i.e., *client* entities), see 8.3.10.1 Finding Matching Publications on page 551.
- Getting *DataReader* protocol or cache statuses, see 8.3.5 Checking DataReader Status and StatusConditions on page 514 and 8.3.7 Statuses for DataReaders on page 516.
- Modifying the QoS.

To access these underlying objects:

- `dds::pub::DataWriter<RequestType> request_datawriter()`
- `dds::sub::DataReader<ReplyType> reply_datareader()`
- `dds::pub::DataWriter<ReplyType> reply_datawriter()`
- `dds::sub::DataReader<RequestType> request_datareader()`

## 28.4 Generating RPC Code from IDL using RTI Code Generator

To use RPC, add the `@DDSService` or `@service` annotation to an interface in an IDL file. For example:

```java
module robot {
    exception TooFast {};
}
```
enum Command { START_COMMAND, STOP_COMMAND };  

struct Status {  
    string msg;  
};  

@DDSService  
interface RobotControl {  
    void command(Command com);  
    float setSpeed(float speed) raises (TooFast);  
    float getSpeed();  
    void getStatus(out Status status);  
};  

This IDL file defines an interface with four methods:  

- `command` receives one argument and returns nothing.  
- `setSpeed` receives one argument, returns a `float`, and can throw a `TooFast` exception.  
- `getSpeed` receives no argument and returns a `float`.  
- `getStatus` receives one output argument and returns nothing.

To generate the supporting code and example client and service applications, run `RTI Code Generator` as follows:

```
rtiddsgen -language C++11 -example <architecture> <IDL file name>.idl
```

This will generate:

- `<IDL file name>.hpp`, `<IDL file name>.cxx`, `<IDL file name>Plugin.hpp`, `<IDL file name>Plugin.cxx`. These files contain the support code for the type defined in the IDL.  
- `<IDL file name>_service.cxx` contains the example service application, and the service implementation.  
- `<IDL file name>_client.cxx` contains the example client application.

For more information and an example, see the RPC Tutorial in the RTI Connext Modern C++ API Reference HTML documentation.
Part 5: RTI Real-Time WAN Transport

Real-Time WAN Transport is a Connext DDS transport plugin that enables communication over wide area networks (WANs) using UDP as the underlying IP transport-layer protocol.

The material in this part of the manual is only relevant if you have installed Real-Time WAN Transport. This feature is not installed as part of a Connext DDS package; it must be downloaded and installed separately. See the RTI Real-Time WAN Transport Installation Guide for details. See also the RTI Real-Time WAN Transport Release Notes.

All new applications that communicate over wide area networks using UDP should use Real-Time WAN Transport. This release includes RTI Secure WAN Transport (described in Part 6: RTI Secure WAN Transport on page 1109) only for compatibility with existing applications, which should upgrade to Real-Time WAN Transport. RTI may not support Secure WAN Transport in future versions of Connext DDS and no longer provides it to new customers.

This section includes:

- Introduction to Real-Time WAN Transport (Chapter 29 on page 1035)
- Transport Capabilities (Chapter 30 on page 1038)
- Communication Scenarios (Chapter 31 on page 1045)
- Deployment Scenarios (Chapter 32 on page 1052)
- Enabling Real-Time WAN Transport (Chapter 33 on page 1064)
- Transport Initial Peers (Chapter 34 on page 1066)
- Transport Configuration (Chapter 35 on page 1068)
- Security (Chapter 36 on page 1086)
- Advanced Concepts (Chapter 37 on page 1087)
- Transport Debugging (Chapter 38 on page 1095)
- Troubleshooting (Chapter 40 on page 1104)
Chapter 29 Introduction to Real-Time WAN Transport

*Real-Time WAN Transport* is a smart transport that enables secure, scalable, and high-performance communication over wide area networks (WANs), including public networks. It extends *Connext DDS* capabilities to WAN environments. *Real-Time WAN Transport* uses UDP as the underlying IP transport-layer protocol to better anticipate and adapt to the challenges of diverse network conditions, device mobility, and the dynamic nature of WAN system architectures.

*Real-Time WAN Transport*, in combination with *RTI Cloud Discovery Service*, provides a complete, seamless solution out of the box for WAN connectivity. This WAN connectivity solution, including *Real-Time WAN Transport* and *Cloud Discovery Service*, is available as an optional add-on.

*Real-Time WAN Transport* replaces the transport capabilities of the *Secure WAN Transport* optionally available with previous *Connext DDS* releases, and provides the following capabilities:

- **NAT (Network Address Translator) traversal**: Ability to communicate between *DomainParticipants* running in a Local Area Network (LAN) that is behind a NAT-enabled router, and *DomainParticipants* on the outside of the NAT across a WAN. This functionality is provided in combination with *Cloud Discovery Service* (see 30.1 NAT Traversal on page 1038).

- **IP mobility**: Support for network transitions and changes in IP addresses in any of the *DomainParticipants* participating in the communication (see 30.2 IP Mobility on page 1043).

- **Security**: Secure communications between *DomainParticipants* using *Security Plugins* (see Chapter 36 Security on page 1086).

*Real-Time WAN Transport* does not require third-party components, such as STUN servers, or protocols like SIP to handle session establishment. Using a single API and security model, you can leverage the extensive capabilities of the *Connext DDS* framework and ecosystem, including tools
and infrastructure services, even for real-time connectivity from edge to cloud and back in highly distributed systems that communicate across wide area networks.

For Real-Time WAN Transport example code, see https://github.com/rticomunity/rticonnextdds-examples/tree/develop/examples/connex_dds/real_time_wan_transport.

29.1 Key Terms

29.1.1 Basic Terms

Wide Area Network (WAN): A wide area network (WAN) is a collection of local area networks (LANs) or other networks that communicate with one another. A WAN is essentially a network of networks, with the Internet being the world's largest WAN.

Cellular Network (or Cellular WAN): A cellular network is a wide area network for voice and data that is typically provided by the cellular carriers to transmit a wireless signal over a range of several miles to a mobile device.

External DomainParticipant: A DomainParticipant using a Real-Time WAN Transport that is publicly reachable at a public address. Being reachable at a public IP address does not mean that the DomainParticipant is not behind a NAT-enabled router. It is possible that an external DomainParticipant is behind a NAT-enabled router if the network administrator configures a static NAT mapping between the DomainParticipant private address and a public address.

29.1.2 IP Address Types

IP Transport Address (or Address): The combination of the IPv4 address and the UDP Port where an application accepts incoming traffic. Sometimes you will also see the term "address" being used to refer to an IP transport address when the context is clear.

External IP Transport Address (or External Address or Public Address): An IP transport address that is routable on a WAN. When the WAN is the Internet, the term "Internet-routable address" can be used instead.

Private IP Transport Address (or Private Address or Internal Address): The IP transport address of an application that sits behind a NAT. This address is not reachable from external applications running outside the NAT.

Service Reflexive Address: The public IP transport address that Cloud Discovery Service obtains for a UUID locator contained in the participant announcement sent by a DomainParticipant.

29.1.3 Locators

RTPS Locator (or Locator): A Connext DDS endpoint (DataWriter or DataReader) address unit that consists of a transport class, RTPS port, and locator transport address (128-bit).
Reachable Locator: Locator associated with a DDS endpoint (DataWriter or DataReader) to which another DDS endpoint can send data.

RTPS UUID WAN Locator (or UUID Locator): A WAN locator for a Real-Time WAN Transport that is not reachable. UUID locators are transformed into UUID+PUBLIC locators by associating a public IP transport address to the UUID.

RTPS UUID+PUBLIC WAN Locator (or UUID+PUBLIC Locator): A WAN locator for a Real-Time WAN Transport that is reachable. The locator encapsulates a public IP transport address as part of the locator address.

29.1.4 WAN Ecosystem

Session Traversal Utilities for NAT (STUN): Standardized set of methods, including a network protocol, for traversal of network address translator gateways in applications of real-time voice, video, messaging, and other interactive communications.

STUN Server: A STUN server enables clients to find out their public IP transport address and NAT type.

Interactive Connectivity Establishment (ICE): ICE is a protocol used for NAT traversal. ICE uses a combination of methods including STUN and Traversal Using Relay NAT (TURN) to traverse NATs.
Chapter 30 Transport Capabilities

30.1 NAT Traversal

In WAN environments, applications running behind a NAT-enabled router typically need to communicate with applications running behind a different NAT-enabled router. NAT (Network Address Translation) is a method of remapping one private IP address and port into a public IP address and port by modifying the IP address and port information in the IP header of the packets while they are in transit across a NAT-enabled router. The technique has become a popular and essential tool in conserving the IPv4 global address space in the face of IPv4 address exhaustion. Many applications with individual private IP addresses can utilize a NAT-enabled router to communicate with external applications using a single public IP address.

Figure 30.1: NAT Traversal

1 Send
Source: 192.168.100.3:3655
Dest: 209.131.36.58:4500

192.168.100.3

192.168.100.4

192.168.100.5

2 Map
Map private IP & port to public IP & port

<table>
<thead>
<tr>
<th>Private IP Addr &amp; Port</th>
<th>NAT Translation Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>192.168.100.3:3655</td>
<td>145.12.131.7:6282</td>
</tr>
</tbody>
</table>

3 Receive
Source: 145.12.131.7:6282
Dest: 209.131.36.58:4500

192.168.1.1

NAT Router

145.12.131.7
30.1.1 NAT Kinds

There are four kinds of NATs:

30.1.1.1 Full-Cone NAT (or One-to-One NAT)

A full-cone NAT, also known as a one-to-one NAT, has the following characteristics:

- Once an internal address (iAddr:iPort) is mapped to an external address (eAddr:ePort) by the NAT router, any packets from (iAddr:iPort) are sent through (eAddr:ePort).
- Any external host can send packets to (iAddr:iPort) by sending packets to (eAddr:ePort), regardless of the external host address/port (dAddr:dPort) used to send the packets.

Figure 30.2: Full-Cone NAT

<table>
<thead>
<tr>
<th>NAT Translation Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(iAddr:iPort)</td>
</tr>
<tr>
<td>192.168.100.3:3855</td>
</tr>
<tr>
<td>192.168.100.3:3855</td>
</tr>
</tbody>
</table>

Real-Time WAN Transport in combination with Cloud Discovery Service will enable communications between Connext DDS applications running between different kinds of NATs. For information on the various NAT kinds, please see the following sections.
30.1.1.2 Address-Restricted-Cone NAT

- Once an internal address (iAddr:iPort) is mapped to an external address (eAddr:ePort) by the NAT router, any packets from (iAddr:iPort) are sent through eAddr:ePort.
- An external host (dAddr:any) can send packets to (iAddr:iPort) by sending packets to (eAddr:ePort) only if (iAddr:iPort) has previously sent a packet to (dAddr:any). "Any" means the port number doesn't matter.

30.1.1.3 Port-Restricted Cone NAT

This NAT is similar to an address-restricted cone NAT, but the restriction also includes port numbers.

- Once an internal address (iAddr:iPort) is mapped to an external address (eAddr:ePort) by the NAT router, any packets from (iAddr:iPort) are sent through (eAddr:ePort).
- An external host (dAddr:dPort) can send packets to (iAddr:iPort) by sending packets to (eAddr:ePort) only if (iAddr:iPort) has previously sent a packet to (dAddr:dPort).

30.1.1.4 Symmetric NAT

- Each request from the same internal IP address and port (iAddr:iPort) to a specific destination IP address and port (dAddr:dPort) is mapped by the NAT router to a unique external source IP address and port (eAddr:ePort). If the same internal host sends a packet even with the same source address and port but to a different destination, a different mapping is used.
- Only an external host that receives a packet from an internal host can send a packet back.
30.1.2 Identifying the NAT Type

There are multiple third-party utilities that you can download to find out the NAT type. One example is natat (https://github.com/songjiayang/natat).

```
> ./natat
2020/11/21 11:10:52 start stun server ping...
2020/11/21 11:10:53 stun1.l.google.com:19302 mapped: 0.0.0.0:3489 -> 99.35.17.233:3489
2020/11/21 11:10:53 stun1.l.google.com:19302 mapped: 0.0.0.0:3489 -> 99.35.17.233:3489
2020/11/21 11:10:53 start NAT type assert...
2020/11/21 11:10:53 It's Cone NAT
```

30.1.3 NAT Bindings

Applications behind a NAT cannot receive data from applications outside the NAT unless they open a UDP NAT binding (or UDP hole) with each one of the public IP transport addresses associated with the applications running outside the NAT.

A NAT binding creates a mapping between a private IP transport address (iAddr:iPort) and a public IP transport address (eAddr:ePort) for a given set of destination IP transport addresses. There are two kinds of bindings:
• **Static bindings (also known as port forwarding):** You can set the configuration of a NAT-enabled router to map (iAddr:iPort) to (eAddr:ePort) (see Figure 30.4: Open a Static Binding in a NAT-Enabled Router below) for all destination addresses. These bindings allow incoming traffic from any external IP transport address.

• **Dynamic bindings:** The bindings are opened dynamically when the application running inside the NAT sends a message to a destination IP transport address outside the NAT. The behavior of the dynamic bindings depends on the type of NAT (see 30.1.1 NAT Kinds on page 1039). Unlike static bindings, dynamic bindings can expire if there is no outgoing traffic (see 30.1.4 NAT Bindings Expiration below).

**Figure 30.4: Open a Static Binding in a NAT-Enabled Router**

![Figure 30.4: Open a Static Binding in a NAT-Enabled Router](image)

30.1.4 NAT Bindings Expiration

*Real-Time WAN Transport* will be able to establish new NAT bindings if the old bindings are closed by the NAT-enabled router without any user intervention. In addition, *Real-Time WAN Transport* provides a built-in **Ping** mechanism to keep the NAT bindings open at all times. This
capability eliminates the latency penalty that is introduced during the process of establishing a new NAT binding.

For security purposes, in the absence of outbound traffic, the NAT binding from an internal address (iAddr:iPort) to an external address (eAddr:ePort) usually expires after periods of time in the range of tens of seconds to a few minutes. When it expires, the NAT binding is removed and it closes. The expiration time can usually be configured (see Figure 30.5: Session Timeout below).

Figure 30.5: Session Timeout

30.1.5 NAT Hairpinning

Real-Time WAN Transport does not require support for NAT hairpinning and can be used in combination with the builtin UDPv4 transport in a DomainParticipant to support both communication with DomainParticipants within the same LAN and communication with DomainParticipants in a WAN simultaneously.

In network computing, hairpinning (or NAT loopback) refers to communication between two hosts behind the same NAT router using their mapped external address (eAddr:ePort). Because not all NAT routers support this communication configuration, usually applications must be designed to be aware of it.

30.2 IP Mobility

Real-Time WAN Transport automatically and transparently handles the IP address changes in the applications communicating over the WAN, without any application intervention.

In WAN communication scenarios, it is common for applications to roam among different networks, changing their IP addresses. For example, assume the following scenario:
Connectivity continues with the vehicle while it transitions between different networks as it drives from one point to another in the city.
Chapter 31 Communication Scenarios

*Real-Time WAN Transport* can be used to address two basic communication scenarios between *Connext DDS DomainParticipants*.

### 31.1 Peer-to-Peer Communication with a Participant that has a Public Address

In this scenario, one or more *DomainParticipants* behind any kind of NAT (including symmetric NATs) communicate with a *DomainParticipant* with a well-known public IP transport address (see Figure 31.1: Peer-to-Peer between a Participant behind Any Kind of NAT and an External Participant on the next page).
31.1 Peer-to-Peer Communication with a Participant that has a Public Address

Figure 31.1: Peer-to-Peer between a Participant behind Any Kind of NAT and an External Participant

Note that Cloud Discovery Service (CDS) is not needed to facilitate NAT traversal in this scenario because the external DomainParticipant can figure out the public IP transport addresses at which the Internal Participant is reachable by looking at the UDP packets coming from the Internal Participant.

The external DomainParticipant must be reachable at a well-known public address (50.10.23.45:2345 in this example). What this means is that the address must be routable on the WAN.

There are two configurations that allow the association of External Participant with a well-known public address:

- **Sub-Scenario 1**: The DomainParticipant is associated directly with the public address 50.10.23.45:2345, or
- **Sub-Scenario 2**: The DomainParticipant is behind a NAT-enabled router in which you have created a static NAT mapping from the DomainParticipant private IP transport address 192.168.1.1:1234 to the public IP transport address 50.10.23.45:2345.
31.1.1 External Participant Configuration: Sub-Scenario 1

For a detailed description of how the communication establishment protocol works in this scenario, see 37.4 Communication Establishment Protocol for Peer-to-Peer Communication with a Participant that has a Public Address on page 1092.

The following section provides an example configuration for the scenario described in Figure 31.1: Peer-to-Peer between a Participant behind Any Kind of NAT and an External Participant on the previous page.

31.1.1 External Participant Configuration: Sub-Scenario 1

```
<dds>
  <qos_profile name="ExternalParticipant">
    <domainParticipant_qos>
      <transport_builtin>
        <mask>UDPv4_WAN</mask>
        <udpv4_wan>
          <public_address>50.10.23.45</public_address>
          <comm_ports>
            <default>
              <host>2345</host>
              <public>2345</public>
            </default>
          </comm_ports>
        </udpv4_wan>
      </transport_builtin>
    </domainParticipant_qos>
  </qos_profile>
</dds>
```

- To enable Real-Time WAN Transport, `<mask>` within `<transport_builtin>` must contain UDPv4_WAN.
- `<public_address>` contains the IP address of the host where the external DomainParticipant is running.
- `<comm_ports>` defines the port ( `<host>`) in which the external DomainParticipant receives/sends data.

31.1.2 External Participant Configuration: Sub-Scenario 2

```
<dds>
  <qos_profile name="ExternalParticipant">
    <domainParticipant_qos>
      <transport_builtin>
        <mask>UDPv4_WAN</mask>
        <udpv4_wan>
          <public_address>50.10.23.45</public_address>
          <comm_ports>
            <default>
              <host>1234</host>
              <public>2345</public>
            </default>
          </comm_ports>
        </udpv4_wan>
      </transport_builtin>
    </domainParticipant_qos>
  </qos_profile>
</dds>
```
To enable *Real-Time WAN Transport*, `<mask>` within `<transport_builtin>` must contain UDPv4 WAN.

- `<public_address>` contains the public IP address in the NAT-enabled router to which the private IP address is mapped.
- `<comm_ports>` defines the mapping between the following ports:
  - `<host>`: local UDP port in which the external *DomainParticipant* receives/sends data in the machine where it is running.
  - `<public>`: public port to which the local UDP port is mapped in the NAT-enabled router.

### 31.1.3 Internal Participants Configuration

```xml
<dds>
    <qos_profile name="InternalParticipant">
        <domain_participant_qos>
            <transport_builtin>
                <mask>UDPv4_WAN</mask>
            </transport_builtin>
            <discovery>
                <initial_peers>
                    <element>0@udpv4_wan://50.10.23.45:2345</element>
                </initial_peers>
            </discovery>
        </domain_participant_qos>
    </qos_profile>
</dds>
```

- To enable *Real-Time WAN Transport*, `<mask>` within `<transport_builtin>` must contain UDPv4 WAN.
- In addition, the *InternalParticipant* must set its initial peers to point to the external *DomainParticipant* public address.

### 31.2 Peer-to-Peer Communication with Participants behind Cone NATs Using Cloud Discovery Service

In this communication scenario, all the *DomainParticipants* communicating over a WAN are behind cone NATs (see *Figure 31.2: Peer-to-Peer between Participants behind Cone NATs on the next page*).

This scenario requires *Cloud Discovery Service* (CDS) to map the private addresses of a *DomainParticipant* into public addresses (called service reflexive addresses) and to provide these public addresses to other *DomainParticipants* so that they can start communicating with the *DomainParticipant* peer-to-peer.
In this scenario, CDS must be reachable at a well-known public address, and the DomainParticipants participating in the communication must include the CDS address in their initial peers.

**Figure 31.2: Peer-to-Peer between Participants behind Cone NATs**

Cloud Discovery Service also serves as a directory, so that a participant only needs to know about the CDS public address to connect to multiple peers automatically.

The service reflexive addresses obtained by CDS are provided as part of the WAN locators contained in the participant announcement sent from CDS to the application DomainParticipants.

In **Figure 31.2: Peer-to-Peer between Participants behind Cone NATs above**, eAddr2:ePort2 is the service reflexive address that CDS obtains for Internal Participant 2, and eAddr1:ePort1 is the service reflexive address that CDS obtains for Internal Participant 1. CDS provides eAddr2:ePort2 to the Internal Participant 1 so that it can reach the Internal Participant 2; it provides eAddr1:ePort1 to the Internal Participant 2 so that it can reach the Internal Participant 1.

For a detailed description of how the communication establishment protocol works in this scenario, see 37.3 Communication Establishment Protocol for Peer-to-Peer Communication with Participants behind Cone NATs on page 1089.

The following section provides an example configuration for the scenario described in **Figure 31.2: Peer-to-Peer between Participants behind Cone NATs above**.
31.2.1 Internal Participants Configuration

```xml
<dds>
  <qos_profile name="InternalParticipant">
    <domain_participant_qos>
      <transport_builtin>
        <mask>UDPv4_WAN</mask>
      </transport_builtin>
    </discovery>
      <initial_peers>
        <element>rtps@udpv4_wan://60.10.23.45:2345</element>
      </initial_peers>
    </discovery>
  </domain_participant_qos>
</qos_profile>
</dds>
```

- To enable the *Real-Time WAN Transport*, specify the transport in `<transport_builtin>/<mask>` as UDPv4_WAN.
- When you specify the initial peers of the InternalParticipant in `<initial_peers>/<element>`, use the public address of the *Cloud Discovery Service*.

31.2.2 Cloud Discovery Service Configuration

```xml
<dds>
  <cloud_discovery_service name="CDS">
    <transport>
      <element>
        <alias>builtin.udpv4_wan</alias>
        <receive_port>2345</receive_port>
      </element>
    </transport>
    <property>
      <element>
        <name>dds.transport.UDPv4_WAN.builtin.public_address</name>
        <value>60.10.23.45</value>
      </element>
    </property>
  </cloud_discovery_service>
</dds>
```

- To enable *Real-Time WAN Transport* in CDS, set `<alias>` to builtin.udpv4_wan.
- `<receive_port>` contains the public UDP port in which CDS is reachable by the DomainParticipants.
- `dds.transport.UDPv4.builtin.public_address` contains the public IP address in which CDS is reachable by the DomainParticipants.

For additional details on each one of the parameters of the CDS instance configuration, see the *RTI Cloud Discovery Service* documentation.
As described above, CDS must be reachable in a well-known public address, in this example 60.10.23.45:2345. A "well-known" public address is an address that is routable on the WAN.

There are two configurations that allow the association of CDS with a well-known public address:

1. The CDS network interface card (NIC) is associated directly with the public address.
2. CDS is behind a NAT-enabled router, and you have created a static NAT mapping from the CDS private address iAddr:iPort to the public address 60.10.23.45:2345.

If CDS is behind a NAT-enabled router, the host port (iPort) must be the same as the <receive_port> (2345). If you want to use a different host port, it will be necessary to configure the property dds.trans-port.UDPv4_WAN.builtin.comm_ports (35.2.1.1 Changing the UDP Port Mapping on page 1079).

As described above, this communication scenario requires that all the DomainParticipants participating in the communication are behind cone NATs. Communication is not possible if any of them is behind a symmetric NAT. Therefore, it is mandatory that you verify the type of NAT in which the applications run. See 30.1.2 Identifying the NAT Type on page 1041.
Chapter 32 Deployment Scenarios

The communication scenarios described in Chapter 31 Communication Scenarios on page 1045 provide the building blocks for WAN communication using Real-Time WAN Transport. However, they do not take into consideration important communication aspects such as scalability.

The intent of this section is to describe some of the most common deployment scenarios for Real-Time WAN Transport.

32.1 Edge-to-Data Center Deployment Scenario

In traditional Connext DDS applications, most of the data processing is done on the edge devices. However, as these applications are distributed across the WAN, it becomes necessary to move some computation and storage to data centers or clouds.

Consider the use case of autonomous driving technology. In this scenario, each vehicle has one or more internal Connext databuses in which different applications run to provide capabilities such as sensor fusion, path planning, vehicle control, and so on.
Some of the information generated in the Edge Connext Dabus may have to be sent outside the vehicle to different data centers and/or clouds to support use cases such as data storage, data analytics, and others. Likewise, the vehicle may have to receive information from the data centers and/or cloud such as over-the-air (OTA) updates.

Instead of configuring a large number of DomainParticipants running inside the vehicle to use Real-Time WAN Transport, it is more scalable and secure to provide a gateway component within the vehicle whose main purpose is to send and receive the necessary information from the data centers. This gateway component is provided by RTI Routing Service.

There is also a gateway Routing Service instance running in the data center in order to send and receive information from the vehicles. That way it is not necessary to configure every DomainParticipant running in the data center to use Real-Time WAN Transport.

Figure 32.2: Edge-to-Data Center Communication on the next page shows the deployment scenario for a fleet of vehicles using a Routing Service in the vehicles and in the data center.
32.1.1 Data Center Routing Service Configuration

The following sections provide an example configuration for the scenario described in Figure 32.2: Edge-to-Data Center Communication above. The Routing Service instances are configured to propagate every Topic. However, in a real scenario only a subset of the Topics would be propagated.

**32.1.1 Data Center Routing Service Configuration**

```xml
<dds>
  <routing_service name="RS">
    <domain_route name="TwoWayDomainRoute">
      <participant name="0">
        <participant_qos>
          <transport_builtin>
            <mask>UDPv4</mask>
          </transport_builtin>
        </participant_qos>
      </participant>
      <participant name="1">
        <participant_qos>
          
        </participant_qos>
      </participant>
    </domain_route>
  </routing_service>
</dds>
```
Participant 0 in the domain route is the DomainParticipant that will be used to send/receive data from the internal Connext databus running in the data center network. The DomainParticipant is configured to use the builtin UDPv4 transport.
Participant 1 is used to send/receive data to/from the WAN, and it has the role of the External Participant described in Sub-scenario 2 in 31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045.

**Note:** By making Participant 1 reachable at a well-known public address 40.20.23.56:5678 (by configuring the data center’s NAT router to do port forwarding), the system doesn’t depend on the kinds of NATs in the vehicles and the rest of the components in the system. Communication is always allowed.

### 32.1.2 In-Vehicle Routing Service Configuration

```xml
<dds>
  <routing_service name="RS">
    <domain_route name="TwoWayDomainRoute">
      <participant name="0">
        <participant_qos>
          <transport_builtin>
            <mask>UDPv4</mask>
          </transport_builtin>
        </participant_qos>
      </participant>
      <participant name="1">
        <participant_qos>
          <transport_builtin>
            <mask>UDPv4_WAN</mask>
          </transport_builtin>
        </participant_qos>
      </participant>
    </domain_route>
    <session name="Session1">
      <auto_topic_route name="AllForward">
        <input participant="0">
          <allow_topic_name_filter>*</allow_topic_name_filter>
          <allow_registered_type_name_filter>*</allow_registered_type_name_filter>
        </input>
        <output participant="1">
          <allow_topic_name_filter>*</allow_topic_name_filter>
          <allow_registered_type_name_filter>*</allow_registered_type_name_filter>
        </output>
      </auto_topic_route>
    </session>
    <session name="Session2">
    </session>
  </routing_service>
</dds>
```
Participant 0 in the domain route is the *DomainParticipant* that will be used to send/receive data from the internal *Connext* databus running in the in-vehicle network. The *DomainParticipant* is configured to use the builtin UDPv4 transport.

Participant 1 is used to send/receive data to/from the WAN and it has the role of the Internal Participant described in Sub-scenario 2 in 31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045.

Notice the use of the ‘0@’ as the participant ID limit in the `<initial_peers>` for Participant 1. This is done to minimize the amount of discovery traffic sent to the *Routing Service* running in the data center. There is no need to use a number different than 0 when using *Real-Time WAN Transport*. (You can, but there will be more traffic.) See 15.2.1 Peer Descriptor Format on page 769 for additional information on the participant ID limit.

### 32.2 Relayed Edge-to-Edge Deployment Scenario

For this deployment scenario, consider a webinar platform built using *Connext DDS*. In this use case, the platform provider does not have any control over the attendees or the presenter network environment, including the NAT configuration. Communication cannot be peer-to-peer for two main reasons:

- The NATs environment is not known in advance.
- Communication is one-to-many. The presenter computer may not have enough resources (CPU and bandwidth) to broadcast the webinar content to all attendees.

To implement this use case, the *Connext DDS* application running in the presenter’s computer will distribute the webinar content to a *Routing Service* instance running in a data center or the cloud. Unlike in the edge-to-data center scenario, where *Routing Service* is used as a gateway distributing information to components that are running inside the cloud, in this scenario *Routing Service* is used as a relay service.
The edge applications cannot communicate with each other directly, so in order to exchange messages, an application sends the message to *Routing Service*, which relays this message to the other applications.

In this case, the *Routing Service* instance will be in charge of relaying the content to the different attendees. Note that, for the sake of simplicity, we only have one *Routing Service* in this example. In a real use case, there may be multiple *Routing Services* organized in a hierarchical manner relaying the signal to different sets of attendees.
32.2 Relayed Edge-to-Edge Deployment Scenario

Figure 32.3: Relayed Edge-to-Edge Communication

Presenter

Attendee 1

Attendee 2

Data Center

Webinar Connext App

RWT

Any NAT

Webinar Connext App

RWT

Any NAT

Webinar Connext App

RWT

Any NAT

RWT = Real-Time WAN Transport

Publicly reachable at 40.20.23.56:5678
The following sections provide an example configuration for the scenario described in Figure 32.3: Relayed Edge-to-Edge Communication on the previous page. The Routing Service instance is configured to propagate every Topic using an AutoTopicRoute.

### 32.2.1 Data Center Routing Service Configuration

```xml
<dds>
  <routing_service name="RS">
    <domain_route name="TwoWayDomainRoute">
      <participant name="1">
        <participant_qos>
          <transport_builtin>
            <mask>UDPv4_WAN</mask>
            <udpv4_wan>
              <public_address>40.20.23.56</public_address>
              <comm_ports>
                <default>
                  <host>4500</host>
                  <public>5678</public>
                </default>
              </comm_ports>
            </udpv4_wan>
          </transport_builtin>
        </participant_qos>
      </participant>
      <session name="Session1">
        <auto_topic_route name="AllForward">
          <input participant="1">
            <allow_topic_name_filter>*</allow_topic_name_filter>
            <allow_registered_type_name_filter>*</allow_registered_type_name_filter>
          </input>
          <output participant="1">
            <allow_topic_name_filter>*</allow_topic_name_filter>
            <allow_registered_type_name_filter>*</allow_registered_type_name_filter>
          </output>
        </auto_topic_route>
      </session>
      <session name="Session2">
        <auto_topic_route name="AllBackward">
          <input participant="1">
            <allow_topic_name_filter>*</allow_topic_name_filter>
            <allow_registered_type_name_filter>*</allow_registered_type_name_filter>
          </input>
          <output participant="1">
            <allow_topic_name_filter>*</allow_topic_name_filter>
            <allow_registered_type_name_filter>*</allow_registered_type_name_filter>
          </output>
        </auto_topic_route>
      </session>
    </domain_route>
  </routing_service>
</dds>
```
The domain route in *Routing Service* only creates one Participant that is used to relay the webinar content and that has the role of the External Participant described in Sub-Scenario 2 in 31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045. The *Routing Service* Participant is reachable at a well-known public IP transport address, 40.20.23.56:5678.

### 32.2.2 Webinar Application Configuration

```xml
<dds>
  <qos_profile name="WebinarParticipant">
    <domain_participant_qos>
      <transport_builtin>
        <mask>UDPv4_WAN</mask>
      </transport_builtin>
      <discovery>
        <initial_peers>
          <element>0@udpv4_wan://40.20.23.56:5678</element>
        </initial_peers>
      </discovery>
    </domain_participant_qos>
  </qos_profile>
</dds>
```

The Participant in the webinar application is used to send/receive webinar data to/from the WAN. This WebinarParticipant has the role of the Internal Participant described in Sub-scenario 2 in 31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045.

Notice the use of the ‘0@’ participant index in the `<initial_peers>`. This is done to minimize the amount of discovery traffic sent to the *Routing Service* running in the data center. There is no need to use a number different than 0 when using *Real-Time WAN Transport*. (You can, but there will be more traffic.) See 15.2.1 Peer Descriptor Format on page 769 for additional information on the participant ID limit.

### 32.3 Peer-to-Peer, Edge-to-Edge Deployment Scenario

Currently, this scenario is only supported in environments in which the NATs involved in the communication are cone NATs. Consider a videoconferencing system in which users establish 1-1 calls with each other. Users communicate with other users within their own LAN and in different LANs across a WAN.
Because each user is behind a cone NAT, the users can communicate peer-to-peer with other users across the WAN with the help of Cloud Discovery Service to facilitate both the NAT traversal process and the discovery of DomainParticipants as described in 31.2 Peer-to-Peer Communication with Participants behind Cone NATs Using Cloud Discovery Service on page 1048.

In addition, some users will be running in the same LAN. Because NAT loopback (see 30.1.5 NAT Hairpinning on page 1043) is not allowed by NAT routers in most cases, it is necessary to use the built-in UDPv4 transport in combination with the Real-Time WAN Transport to enable communications within the LAN.
The following sections provide an example configuration for the scenario described in Figure 32.4: Peer-to-Peer and Edge-to-Edge Communication on the previous page.

### 32.3.1 Video Connext Application Configuration

```
<dds>
  <qos_profile name="VideoAppParticipant">
    <domain_participant_qos>
      <transport_builtin>
        <mask>UDPv4|UDPv4_WAN</mask>
      </transport_builtin>
      <discovery>
        <initial_peers>
          <element>udpv4://239.255.0.1</element>
          <element>rtps@udpv4_wan://40.20.23.56:5678</element>
        </initial_peers>
      </discovery>
      <discovery_config>
        <locator_reachability_assert_period>
          <sec>15</sec>
        </locator_reachability_assert_period>
        <locator_reachability_lease_duration>
          <sec>60</sec>
        </locator_reachability_lease_duration>
      </discovery_config>
    </domain_participant_qos>
  </qos_profile>
</dds>
```

Within the LAN, discovery is configured to be done over multicast by setting `<initial_peers>` to `udpv4://239.255.0.1`. Over the WAN, discovery will occur using *Cloud Discovery Service*.

### 32.3.2 Cloud Discovery Service Configuration

```
<dds>
  <cloud_discovery_service name="CDS">
    <transport>
      <element>
        <alias>builtin.udpv4_wan</alias>
        <receive_port>5678</receive_port>
      </element>
      <property>
        <element>
          <name>dds.transport.UDPv4.builtin.public_address</name>
          <value>50.10.23.45</value>
        </element>
      </property>
    </transport>
  </cloud_discovery_service>
</dds>
```
Chapter 33 Enabling Real-Time WAN Transport

To use Real-Time WAN Transport, you have two options:

- **(Recommended for all platforms that support dynamic library loading)** Automatic dynamic load of the transport library. This approach only requires you to make sure the Real-Time WAN Transport dynamic Release or Debug library is available in your library path. See the RTI Connext DDS Core Libraries Platform Notes for the platforms that support dynamic Real-Time WAN Transport libraries. See 33.1 Dynamically Loading the Real-Time WAN Transport below.

- Manual link against the applicable transport library. See the RTI Connext DDS Core Libraries Platform Notes for a list of the Real-Time WAN Transport libraries available for your platform. See 33.2 Linking the Real-Time WAN Transport against your Application on the next page for details.

**Note:** Normally, you cannot mix static and dynamic libraries. (See 10.3.2 Mixing Static and Dynamic Libraries not Supported on page 682.) For Real-Time WAN Transport, however, it is recommended that you load the library dynamically, regardless of how you load your core libraries. There is one exception: if your platform does not support dynamic loading, follow the instructions in 33.2 Linking the Real-Time WAN Transport against your Application on the next page for more information on how to link your application against the corresponding Real-Time WAN Transport library.

### 33.1 Dynamically Loading the Real-Time WAN Transport

The recommended way to use the transport is to let Connext DDS automatically load the Real-Time WAN Transport dynamic library. (Not all platforms support dynamic loading. See the RTI Connext DDS Core Libraries Platform Notes for details.)
To allow Connext DDS to load the Real-Time WAN Transport dynamic library, simply make sure that the applicable Real-Time WAN Transport dynamic Release or Debug library is available in your system library search path (e.g., LD_LIBRARY_PATH in Linux systems, PATH on Windows systems, DYLD_LIBRARY_PATH on macOS systems).

### 33.2 Linking the Real-Time WAN Transport against your Application

If the method described in 33.1 Dynamically Loading the Real-Time WAN Transport on the previous page is not available in your system (because your architecture does not support dynamic library loading), you can still use the Real-Time WAN Transport by linking your application against the transport library.

Compared with dynamic loading, you need to pay attention to two things.

First, include the Real-Time WAN Transport library in the list of libraries used during your application linking. See the RTI Connext DDS Core Libraries Platform Notes for the specific library to link for your target platform.

Second, manually tell Connext DDS the pointer to the function of the entry point of the Real-Time WAN Transport library before you create the DomainParticipant. Setting this pointer requires setting the `dds.transport.UDPv4_WAN.builtin.plugin_enabled_function_ptr` property. (See 35.1 Setting Real-Time WAN Transport Properties on page 1068.) Here is an example of how to set this pointer in code:

```c
/* Include the symbol for NDDS_Transport_UDP_WAN_Library_is_plugin_enabled */
#include "transport/transport_udp_wan_library.h"

/* The property name "dds.transport.UDPv4_WAN.builtin.plugin_enabled_function_ptr"
 * indicates the entry point for the Real-Time WAN Transport library.
 * The value MUST be the stringified value of the function pointer of
 * NDDS_Transport_UDP_WAN_Library_is_plugin_enabled. Note that
 * add_pointer_property() API will automatically convert the
 * function pointer to a string.
 */
if (DDS_PropertyQosPolicyHelper_add_pointer_property(
    &participantQos.property,
    "dds.transport.UDPv4_WAN.builtin.plugin_enabled_function_ptr",
    (void *) NDDS_Transport_UDP_WAN_Library_is_plugin_enabled)
    != DDS_RETCODE_OK) {
    /* error */
}
```
Chapter 34  Transport Initial Peers

The initial peers (see 9.5.2.2 Setting the ‘Initial Peers’ List on page 640) for Real-Time WAN Transport have the following form:

0 @ udpv4_wan :// <peer_public_IP_address> <peer_public_port>

Notice that the participant ID limit should always be ‘0@’, since there can be only one DomainParticipant associated with the public IP transport address <peer_public_IP_address>:<peer_public_port>. Note that a participant ID limit different than 0 will be accepted, but it will generate more discovery traffic than necessary.

<peer_public_IP_address>:<peer_public_port> defines the public IP transport address at which the remote DomainParticipant is reachable. For example, assume the following configuration for a remote DomainParticipant:

```
<dds>
  <qos_profile name="ExternalParticipant">
    <domain_participant_qos>
      <transport_builtin>
        <mask>UDPv4_WAN</mask>
        <udp4_wan>
          <public_address>50.10.23.45</public_address>
          <comm_ports>
            <default>
              <host>1234</host>
              <public>2345</public>
            </default>
          </comm_ports>
        </udp4_wan>
      </transport_builtin>
    </domain_participant_qos>
  </qos_profile>
</dds>
```

The initial peer that can be used to establish communication with the remote DomainParticipant is:
Chapter 34  Transport Initial Peers

For scenarios in which *Cloud Discovery Service* (CDS) is involved, the initial peers have the following form:

```
0 @ udpv4_wan :// 50.10.23.45 2345
```

For example, assume the following CDS configuration:

```
<dds>
  <cloud_discovery_service name="CDS">
    <transport>
      <element>
        <alias>builtin.udpv4_wan</alias>
        <receive_port>5678</receive_port>
        <property>
          <element>
            <name>dds.transport.UDPv4.builtin.public_address</name>
            <value>50.10.23.45</value>
          </element>
        </property>
      </element>
    </transport>
  </cloud_discovery_service>
</dds>
```

The initial peer that can be used to establish communication with CDS is:

```
rtps @ udpv4_wan :// 50.10.23.45 5678
```
Chapter 35  Transport Configuration

*Real-Time WAN Transport* is a transport plugin that can be configured in three different ways:

- Programmatically by calling `set_builtin_transport_property()` (see 16.5 Setting Builtin Transport Properties of Default Transport Instance—get/set_builtin_transport_properties() on page 799).
- By specifying predefined property strings in the *DomainParticipant’s* PropertyQosPolicy (see 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy on page 801).
- By using the tag `<domain_participant_qos>/<transport_builtin>/udp4_wan>` in the XML configuration.

### 35.1 Setting Real-Time WAN Transport Properties

Table 35.1 Properties for Real-Time WAN Transport describes the configuration parameters for *Real-Time WAN Transport*:
### Table 35.1 Properties for Real-Time WAN Transport

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udp4_wan&gt;</code>)</th>
<th>Property Name (prefix with <code>dds.transport.UDPv4_WAN.builtin.</code>)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Transport Properties</td>
<td>parent.gather_send_buffer_count_max</td>
<td>Specifies the maximum number of buffers that Connext DDS can pass to the <code>send()</code> method of a transport plugin. The transport plugin <code>send()</code> API supports a gather-send concept, where the <code>send()</code> call can take several discontiguous buffers, assemble and send them in a single message. This enables Connext DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer. However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connext DDS from trying to gather too many buffers into a single <code>send</code> call for the transport plugin. Connext DDS requires all transport-plugin implementations to support a gather-send of at least a minimum number of buffers. This minimum number is <code>NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN</code>. See 16.6.1 Setting the Maximum Gather-Send Buffer Count for UDP Transports on page 819. Default: 16</td>
</tr>
<tr>
<td></td>
<td>parent.message_size_max</td>
<td>The maximum size of a message in bytes that can be sent or received by the transport plugin. Above this size, DDS-level fragmentation will occur. See 23.3 Large Data Fragmentation on page 970. This value must be set before the transport plugin is registered, so that Connext DDS can properly use the plugin. Default for Integrity platforms: 9216 Default for non-Integrity platforms: 65507</td>
</tr>
</tbody>
</table>
35.1 Setting Real-Time WAN Transport Properties

<table>
<thead>
<tr>
<th>XML tag (under &lt;udpv4_wan&gt;)</th>
<th>Property Name (prefix with 'dds.transport.UDPV4_WAN.builtin.')</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;allow_interfaces_list&gt;</td>
<td>parent.allow_interfaces_list</td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. As a property value, interfaces must be specified as comma-separated strings, with each comma delimiting an interface. In XML, they are provided as a set of elements (&lt;element&gt;) under &lt;allow_interfaces_list&gt;. For example, the following are acceptable strings: 192.168.1.1 192.168.1.* 192.168.* ether0 If the list is non-empty, this &quot;white&quot; list is applied before the parent.deny_interfaces_list list. The DomainParticipant will use the resulting list of interfaces to inform its remote participant(s) about which unicast addresses may be used to contact the DomainParticipant. The resulting list restricts reception to a particular set of interfaces for unicast UDP. You must manage the memory of the list. The memory may be freed after the DomainParticipant is deleted. Default: empty list that represents all available interfaces</td>
</tr>
<tr>
<td>&lt;deny_interfaces_list&gt;</td>
<td>parent.deny_interfaces_list</td>
<td>A list of strings, each identifying a range of interface addresses or an interface name. If the list is non-empty, deny the use of these interfaces. As a property value, interfaces must be specified as comma-separated strings, with each comma delimiting an interface. In XML, they are provided as a set of elements (&lt;element&gt;) under &lt;deny_interfaces_list&gt;. For example, the following are acceptable strings: 192.168.1.1 192.168.1.* 192.168.* ether0 This &quot;black&quot; list is applied after the parent.allow_interfaces_list and filters out the interfaces that should not be used for receiving data. The resulting list restricts reception to a particular set of interfaces for unicast UDP. You must manage the memory of the list. The memory may be freed after the DomainParticipant is deleted. Default: empty list that represents no deny interfaces</td>
</tr>
</tbody>
</table>
### Setting Real-Time WAN Transport Properties

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udpv4_wan&gt;</code>)</th>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;properties_bitmap&gt;</code></td>
<td>parent.properties_bitmap</td>
<td>A bitmap that defines various properties of the transport to the Connext DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connext DDS tries to receive a message using the plugin. This is in support of a zero-copy interface. Default: 0</td>
</tr>
<tr>
<td>N/A</td>
<td>property_validation_action</td>
<td>By default, property names given in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the transport:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VALIDATION_ACTION_SKIP: skip validation.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail.</td>
</tr>
<tr>
<td><code>&lt;thread_name_prefix&gt;</code></td>
<td>thread_name_prefix</td>
<td>If you do not set this field, Connext DDS creates the following prefix: ’r’ + ’Tr’ + participant identifier + ‘0’ Where ’r’ indicates this is a thread from RTI, ’Tr’ indicates the thread is related to a transport, and participant identifier contains 5 characters as follows:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If <code>participant_name</code> is set: The participant identifier will be the first 3 characters and the last 2 characters of the <code>participant_name</code>.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If <code>participant_name</code> is not set, then the identifier is computed as <code>domain_id</code> (3 characters) followed by <code>participant_id</code> (2 characters).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If <code>participant_name</code> is not set and the <code>participant_id</code> is set to -1 (default value), then the participant identifier is computed as the last 5 digits of the <code>rtps_instance_id</code> in the participant GUID.</td>
</tr>
</tbody>
</table>
### General UDP Properties

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udp4_wan&gt;</code>)</th>
<th>Property Name (prefix with 'dds.transport.UDPv4_WAN.builtin.')</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;protocol_overhead_max&gt;</code></td>
<td>protocol_overhead_max</td>
<td>Maximum size in bytes of protocol overhead, including headers. This value is the maximum size, in bytes, of protocol-related overhead. Normally, the overhead accounts for UDP and IP headers. The default value is set to accommodate the most common UDP/IP header size. Note that when <code>parent.message_size_max</code> plus this overhead is larger than the UDPv4 maximum message size (65535 bytes), the middleware will automatically reduce the effective <code>message_size_max</code> to 65535 minus this overhead. Default: 28</td>
</tr>
<tr>
<td><code>&lt;send_socket_buffer_size&gt;</code></td>
<td>send_socket_buffer_size</td>
<td>Size in bytes of the send buffer of a socket used for sending. On most operating systems, <code>setsockopt()</code> will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to <code>parent.message_size_max</code>. The maximum value is operating system-dependent. If -1, <code>setsockopt()</code> (or equivalent) will not be called to size the send buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td><code>&lt;recv_socket_buffer_size&gt;</code></td>
<td>recv_socket_buffer_size</td>
<td>Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, <code>setsockopt()</code> will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to <code>parent.message_size_max</code>. The maximum value is operating system-dependent. If -1, <code>setsockopt()</code> (or equivalent) will not be called to size the receive buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
</tbody>
</table>
### 35.1 Setting Real-Time WAN Transport Properties

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udp4_wan&gt;</code>)</th>
<th>Property Name (prefix with ‘dds.transport.UDPv4_WAN.builtin.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
</table>
| `<ignore_loopback_interface>` | ignore_loopback_interface | Prevents the transport plugin from using the IP loopback interface. Three values are allowed:  
  - 0: Forces local traffic to be sent over loopback, even if a more efficient transport (such as shared memory) is installed (in which case traffic will be sent over both transports).  
  - 1: Disables local traffic via this plugin. The IP loopback interface will not be used, even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient transport (such as shared memory) instead of the IP loopback.  
  - -1: Automatic. Enables local traffic via this plugin. To avoid redundant traffic, Connext DDS will selectively ignore the loopback destinations that are also reachable through shared memory.  
  Default: -1 |
| DEPRECATED | ignore_nonup_interfaces | This property is only supported on Windows platforms with statically configured IP addresses.  
  It allows/disallows the use of interfaces that are not reported as UP (by the operating system) in the UDPv4_WAN transport. Two values are allowed:  
  - 0: Allow interfaces that are reported as DOWN.  
    - Setting this value to 0 supports communication scenarios in which interfaces are enabled after the participant is created. Once the interfaces are enabled, discovery will not occur until the participant sends the next periodic announcement (controlled by the parameter `participant_qos.discovery_configparticipant_liveliness_assert_period`).  
    - To reduce discovery time, you may want to decrease the value of `participant_liveliness_assert_period`. For the above scenario, there is one cave: non-UP interfaces must have a static IP assigned.  
  - 1: Do not allow interfaces that are reported as DOWN.  
  Default: 1 |
## 35.1 Setting Real-Time WAN Transport Properties

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udp4_wan&gt;</code>)</th>
<th>Property Name (prefix with <code>dds.transport.UDPv4_WAN.builtin.</code>)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;ignore_nonrunning_interfaces&gt;</code></td>
<td>ignore_nonrunning_interfaces</td>
<td>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean &quot;all resources are allocated&quot;, and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>By default this property is set to 1, so Connext DDS will ignore non-running interfaces.</td>
</tr>
<tr>
<td>DEPRECATED</td>
<td>DEPRECATED</td>
<td>Prevents the transport plugin from doing a zero copy. By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometimes tax the OS resources in a manner that cannot be overcome by the application. The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the hardware, device driver, or the OS to allow the zero-copy feature to work for your application, you may have no choice but to turn off zero-copy. By default this is set to 0, so Connext DDS will use the zero-copy API if offered by the OS.</td>
</tr>
<tr>
<td>N/A</td>
<td>no_zero_copy</td>
<td>Controls the blocking behavior of send sockets. <strong>CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.</strong> Currently two values are defined:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 1 (NDDS_TRANSPORT_UDP_BLOCKING_ALWAYS): Sockets are blocking (default socket options for operating system).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• 0 (NDDS_TRANSPORT_UDP_BLOCKING_NEVER): Sockets are modified to make them non-blocking. <strong>This may cause significant performance problems.</strong></td>
</tr>
<tr>
<td><code>&lt;send_blocking&gt;</code></td>
<td>send_blocking</td>
<td>Default: 1</td>
</tr>
</tbody>
</table>

Real-Time WAN transport properties control the transport plugin’s behavior related to network interfaces and socket blocking. These properties are crucial for ensuring optimal performance and maintaining system stability in networked environments.
## 35.1 Setting Real-Time WAN Transport Properties

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udpV4_wan&gt;</code>)</th>
<th>Property Name (prefix with <code>dds.transport.UDPv4_WAN.builtin.</code>)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;transport_priority_mask&gt;</code></td>
<td>transport_priority_mask</td>
<td>Sets the mask for the transport priority field. This is used in conjunction with <code>transport_priority_mapping_low</code> and <code>transport_priority_mapping_high</code> to define the mapping from the 7.5.26 TRANSPORT_PRIORITY QoSPolicy on page 458 to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket. For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high. If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets. Default: 0</td>
</tr>
<tr>
<td><code>&lt;transport_priority_mapping_low&gt;</code></td>
<td>transport_priority_mapping_low</td>
<td>Sets the low and high values of the output range to IPv4 TOS. These values are used in conjunction with <code>transport_priority_mask</code> to define the mapping from the 7.5.26 TRANSPORT_PRIORITY QoSPolicy on page 458 to the IPv4 TOS field. Defines the low and high values of the output range for scaling. Note that IPv4 TOS is generally an 8-bit value. Default: 0 for <code>transport_priority_mapping_low</code> and 0xFF for <code>transport_priority_mapping_high</code></td>
</tr>
<tr>
<td><code>&lt;transport_priority_mapping_high&gt;</code></td>
<td>transport_priority_mapping_high</td>
<td>This property specifies whether to send a PING message before commencing the discovery process. On certain operating systems or with certain switches the initial UDP packet, configuring the ARP table, was unfortunately dropped. To avoid dropping the initial RTPS discovery sample, a PING message is sent to preconfigure the ARP table in those environments. Default: 1</td>
</tr>
<tr>
<td><code>&lt;send_ping&gt;</code></td>
<td>send_ping</td>
<td>This property specifies whether the UDP checksum will be computed. On Windows and Linux systems, the UDP checksum will not be set when <code>use_checksum</code> is set to 0. This is useful when RTPS protocol statistics related to corrupted messages need to be collected through the operation <code>get_participant_protocol_status()</code> (see 9.3.14 Getting Participant Protocol Status on page 631). Default: 1</td>
</tr>
<tr>
<td><code>&lt;use_checksum&gt;</code></td>
<td>use_checksum</td>
<td></td>
</tr>
</tbody>
</table>

1075
### 35.1 Setting Real-Time WAN Transport Properties

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udp4_wan&gt;</code>)</th>
<th>Property Name (prefix with 'dds.transport.UDPv4_WAN.builtin.')</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IP Mobility Properties</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;interface_poll_period&gt;</code></td>
<td>interface_poll_period</td>
<td>Specifies the period in milliseconds to query for changes in the state of all the interfaces. When possible, the detection of an IP address changes is done asynchronously using the APIs offered by the underlying OS. If there is no mechanism to do that, the detection will use a polling strategy where the polling period can be configured by setting this property. Default: 500</td>
</tr>
<tr>
<td><code>&lt;force_interface_poll_detection&gt;</code></td>
<td>force_interface_poll_detection</td>
<td>This property forces the interface tracker to use a polling method to detect changes to the network interfaces in IP mobility scenarios. It only applies to operating systems that support asynchronous notifications of interface changes. If set to TRUE, the interface tracker will use a polling method that queries the interfaces periodically to detect the changes. If set to FALSE, the interface tracker will use the operating system’s default method. Basically, this property allows you—for an operating system that supports asynchronous notification—to use the polling method instead. Default: FALSE</td>
</tr>
<tr>
<td><code>&lt;disable_interface_tracking&gt;</code></td>
<td>disable_interface_tracking</td>
<td>Enables detection of network interface changes. By default, network interfaces changes are propagated in the form of locators to other applications. This is done to support IP mobility scenarios. For example, you could start an application with Wi-Fi and move to a wired connection. In order to continue communicating with other applications, this interface change must be propagated. You can disable the notification and propagation of interface changes by setting this property to 1.</td>
</tr>
<tr>
<td><strong>WAN Properties</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;public_address&gt;</code></td>
<td>public_address</td>
<td>Public IP address associated with the transport instantiation. The address is the public IP address of the NAT-enabled router that provides access to the WAN. Setting the public IP address is only necessary for the Real-Time WAN Transport associated with an external DomainParticipant in order to support the communication scenario described in 31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045. When this property is set, the DomainParticipant will announce PUBLIC+UUID locators to other DomainParticipants. These locators are reachable locators because they contain a public IP transport address for the DomainParticipant. For additional information on Real-Time WAN Transport locators, see 37.1 Transport Locators on page 1087. By default, the public address is not set.</td>
</tr>
</tbody>
</table>
### 35.1 Setting Real-Time WAN Transport Properties

<table>
<thead>
<tr>
<th>XML tag (under <code>&lt;udp4_wan&gt;</code>)</th>
<th>Property Name (prefix with <code>dds.transport.UDPv4_WAN.builtin.</code>)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;binding_ping_period&gt;</code></td>
<td>binding_ping_period</td>
<td>Configures the period in milliseconds at which BINDING_PING messages are sent by a local transport instance to a remote transport instance. For example, 1000 means to send BINDING_PING messages every second. BINDING_PING messages are used on the sending side to open NAT bindings from a local transport instance to a remote transport instance and they are sent periodically to keep the bindings open. For additional information on the role of BINDING_PING messages opening NAT bindings, see 37.3 Communication Establishment Protocol for Peer-to-Peer Communication with Participants behind Cone NATs on page 1089. On the receiving side, BINDING_PINGS are used to calculate the public IP transport address of a UUID locator. This address will be used to send data to the locator. For additional information on the role of BINDING_PING to associate UUID locators to public IP transport addresses, see 37.4 Communication Establishment Protocol for Peer-to-Peer Communication with a Participant that has a Public Address on page 1092. From a configuration point of view, and to avoid communication disruptions, the period at which a transport instance sends BINDING_PING messages should be smaller than the NAT binding session timeout. This timeout depends on the NAT router configuration. Default: 1000 (1 sec)</td>
</tr>
</tbody>
</table>
| `<port_offset>`               | port_offset                                                   | This property allows using the builtin UDPv4 transport and the Real-Time WAN Transport at the same time.  

```xml
<transport_builtin>
  <mask>UDPv4_WAN|UDPv4</mask>
</transport_builtin>
```

When the UDP ports used by Real-Time WAN Transport are not explicitly set, they are calculated as follows: RTPS port + port_offset. See 35.2 Managing UDP Ports Used for Communication on the next page for additional details. Default: 125 |
| `<comm_ports>`                | comm_ports                                                    | Configures the public and private RTPS ports that a transport instance uses to receive/send RTPS data. See 35.2 Managing UDP Ports Used for Communication on the next page for additional details. If this property is not set (default), the UDP ports used for communications will be derived from the RTPS ports associated with the locators for the DomainParticipant and its endpoints (DataWriters and DataReaders). |
### 35.2 Managing UDP Ports Used for Communication

#### 35.2.1 Receiving Data

By default, *Real-Time WAN Transport* uses one UDP port per RTPS port to receive data. The UDP port number is calculated as RTPS port + port_offset.

A *DomainParticipant* uses two RTPS ports, one for discovery and one for user data. Therefore, *Real-Time WAN Transport* uses two UDP ports out-of-the-box. For information on how the RTPS ports are obtained, see [9.5.9.2 Ports Used for Discovery on page 672](#).

You can also configure a specific *DataWriter* and *DataReader* to receive unicast data in a different RTPS port by configuring the [7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460](#). This will also lead to the usage of a different UDP port by *Real-Time WAN Transport*.

There are two main use cases in which the default mapping from RTPS ports to UDP ports is not suitable:

- The first use case involves the configuration of the External Participant described in [31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045](#). In this use case, you must be able to select the private and public UDP ports used for communication because you have to create a static NAT binding on the router for the External Participant.

- The second use case involves the use of UDP load balancers. With UDP load balancers, you must be able to configure a single UDP port to handle all data reception because the load balancer would not know how to map different ports to the same *DomainParticipant*.

For these use cases, *Real-Time WAN Transport* provides a way to specify the private and public UDP ports that will be used to serve specific RTPS ports.
35.2.1 Changing the UDP Port Mapping

The `<comm_ports>` XML tag or the property `dds.transport.UDPv4_WAN.builtin.comm_ports` can be used to change the mapping of UDP ports to RTPS ports.

You can specify a list of mappings from an RTPS port to a host, and (optionally) a public UDP port. For RTPS ports that are not part of the list, you can provide a default mapping.

When the property `dds.transport.UDPv4_WAN.builtin.comm_ports` is used instead of XML, the list is a JSON string.

35.2.1.2 Configuring the Transport to Use a Single Port for an External Participant behind a NAT

This configuration will be needed for the External Participant behind a NAT in the scenario described in 31.2 Peer-to-Peer Communication with Participants behind Cone NATs Using Cloud Discovery Service on page 1048.

**Figure 35.1: Single Port External Participant**

XML:
<udpv4_wan>
    <comm_ports>
        <default>
            <host>1234</host>
            <public>2345</public>
        </default>
    </comm_ports>
</udpv4_wan>

Property dds.transport.UDPv4_WAN.builtin.comm_ports:
{
    "default": {
        "host": 1234,
        "public": 2345
    }
}

35.2.2 Configuring the Transport to Use a Single Port for an Internal Participant behind a NAT

For the Internal Participants behind NATs used in the scenarios described in Chapter 31 Communication Scenarios on page 1045, it is not necessary to configure the public port. The public port will be automatically assigned by the NAT once packages are sent from the private address.
35.2.2 Configuring the Transport to Use a Single Port for an Internal Participant behind a NAT

Figure 35.2: Single Port Internal Participant

XML:

```xml
<udp4_wan>
  <comm_ports>
    <default>
      <host>1234</host>
    </default>
  </comm_ports>
</udp4_wan>
```

Property `dds.transport.UDPv4_WAN.builtin.comm_ports`:

```json
{
  "default": {
    "host": 1234,
  }
}
```

The mapping between the private address 192.168.1.1:1234 and the public address 40.10.23.45:8900 is automatically created by the NAT-enabled device when the Internal Participant sends a packet out.
35.2.3 Configuring the Transport to Segregate Traffic for a Topic in its own Port

In some cases, you may want to segregate the RTPS traffic for a Topic, such as a Video Topic, in its own port. This Topic will get its own socket and receive a socket buffer. It will also get its own receive thread, which will make data reception on the Topic completely concurrent. For details on the middleware threads, see Chapter 21 Connext DDS Threading Model on page 908.

Figure 35.3: Traffic Segregation in Different Port

35.2.3.1 External Participant Configuration

```xml
<dds>
  <qos_profile name="ExternalParticipant">
    <domain_participant_qos>
      <transport_builtin>
        <mask>UDPv4_WAN</mask>
        <udpv4_wan>
          <public_address>50.10.23.45</public_address>
          <comm_ports>
            <default>
              <host>1234</host>
              <public>2345</public>
            </default>
            <mappings>
              <element>
                <rtps>5001</rtps>
                <host>5000</host>
                <public>3456</public>
              </element>
            </mappings>
          </comm_ports>
        </udpv4_wan>
      </transport_builtin>
    </domain_participant_qos>
  </qos_profile>
</dds>
```
35.2.4 Sending Data

Data is always sent from a single UDP port. There is no way to send data using different UDP ports for different Topics.

The UDP port used for sending data corresponds to the port associated with the discovery RTPS port according to the rules described in 35.2.1 Receiving Data on page 1078. When the <comms_port>/default> is defined, the port used for sending data is the one provided in <comms_port>/default.

To use a different port for the Video Topic, you will have to first change the 7.5.28 TRANSPORT_UNICAST QosPolicy (DDS Extension) on page 460 to specify an RTPS port (<unicast>/<receiver_port>) for video data reception. Then, you will have to configure the mapping to UDP ports by updating the comm_ports configuration.

If you choose to configure the comm_ports using the property dds.transport.UDPv4_WAN.builtin.comm_ports, the following example will be the JSON string for the scenario described in Figure 35.3: Traffic Segregation in Different Port on the previous page:

```json
{
    "default": {
        "host": 1234,
        "Public": 2345
    },
    "mappings": [
        {
            "rtps": 5001,
            "Host": 5000,
            "Public": 3456
        }
    ]
}
```
35.3 Disabling IP Fragmentation for Real-Time WAN Transport

For WAN communications, it is not a good idea to rely on IP fragmentation. IP fragmentation causes significant issues in UDP, where there is no integrated support for a path MTU (maximum transmission unit) discovery protocol as there is in TCP. These are some of the problems associated with IP fragmentation:

- To successfully reassemble a packet, all fragments must be delivered. If a fragment is lost, the whole packet will be lost.
- Before reassembly, a host must hold partial fragment datagrams in memory. This opens an opportunity for memory exhaustion attacks.
- Subsequent fragments lack the higher-layer header. The TCP or UDP header is only present in the first fragment, making it impossible for firewalls to filter fragment datagrams based on criteria like source or destination ports.

For more information on IP-level versus Connext DDS-level fragmentation, see 23.3 Large Data Fragmentation on page 970.

This section describes how to disable IP fragmentation in Connext DDS applications using the RTI Real-Time WAN Transport. Instead, Connext DDS will be responsible for fragmentation, done at the RTPS level. The key changes involve:

- Setting the Real-Time WAN Transport MTU (<message_size_max>) to be smaller than the typical IP MTU of around 1500 bytes. The recommendation is to be even more conservative and set the transport MTU to 1400 bytes.
- Enabling DDS fragmentation for reliable Topics (user and built-in Topics) by configuring the 7.5.20 PUBLISH_MODE QosPolicy (DDS Extension) on page 444.
- Adjusting the RTPS protocol overhead without security to a number that maximizes the user payload ratio per RTPS packet.

For example:

```xml
<qos_profile name="Transport.UDP.WAN">
  <participant_qos>
    <discovery_config>
      <publication_writer_publish_mode>
        <kind>ASYNCHRONOUS_PUBLISH_MODE_QOS</kind>
      </publication_writer_publish_mode>
      <subscription_writer_publish_mode>
        <kind>ASYNCHRONOUS_PUBLISH_MODE_QOS</kind>
      </subscription_writer_publish_mode>
      <secure_volatile_writer_publish_mode>
        <kind>ASYNCHRONOUS_PUBLISH_MODE_QOS</kind>
      </secure_volatile_writer_publish_mode>
      <service_request_writer_publish_mode>
        <kind>ASYNCHRONOUS_PUBLISH_MODE_QOS</kind>
      </service_request_writer_publish_mode>
    </discovery_config>
  </participant_qos>
</qos_profile>
```
Note: Batching does not currently support RTPS fragmentation. If you use batching, you will currently not be able to take advantage of Connext DDS fragmentation. This means that your batch size has to be limited to the transport MTU, which is equal to the <message_size_max>, minus the dds.participant.protocol.rtps_overhead, minus the security rtps_overhead. See 23.3 Large Data Fragmentation on page 970 for more information.

35.3.1 RTPS Overhead

Connext DDS adds protocol information to every RTPS message it sends out. If DDS-level fragmentation is used, the amount of overhead reserved per RTPS message for this protocol information may become relatively large. The property dds.participant.protocol.rtps_overhead (see 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439), with a default value of 512, is used to adjust the overhead of the RTPS protocol without including security. The security overhead added by enabling sub-message or message encryption is automatically added to the dds.participant.protocol.rtps_overhead value.

The maximum user payload per data packet is (<message_size_max> - rtps_overhead - security_rtps_overhead). Therefore, for small transport MTUs (e.g., 1400 bytes), it is recommended to reduce the rtps_overhead value to increase the user payload utilization ratio per packet.

For WAN communications, the recommendation is to use 196 bytes for rtps_overhead. (If that is not enough for your use case, you will see a "serialize buffer too small" error. See 40.4 "serialize buffer too small" Error on page 1106 for information on how to fix the problem.)
Chapter 36 Security

Fine-grained security and access control at the Topic level is provided through the use of RTI Security Plugins, which are the Connext DDS implementation of the OMG 'DDS Security' specification, version 1.1, builtin plugins. For detailed information on how to secure your Connext DDS system, see the RTI Security Plugins User's Manual.

In addition, you can use symmetric cryptography using pre-shared keys to protect the integrity of the Binding Ping messages (see 37.2 Binding Ping Messages on page 1088) and the communication with Cloud Discovery Service. For further details, see the "Support for RTI Real-Time WAN Transport" chapter in the RTI Security Plugins User's Manual.
Chapter 37 Advanced Concepts

37.1 Transport Locators

This section provides information about the format of the locators associated with the *Real-Time WAN Transport*. For general information about RTPS locators, see Chapter 17 RTPS Locators and IP Mobility on page 827.

An RTPS locator is an address at which a DDS endpoint (*DataWriter* or *DataReader*) can be reached. Default locators for discovery endpoints and user data endpoints are exchanged with the Participant Announcement (PA).

An RTPS locator consists of a transport Class ID, an address of 128 bits, and a logical port called the RTPS port, as shown in Figure 37.1: RTPS Locator below.

**Figure 37.1: RTPS Locator**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Long class_id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDSUnsignedLong rtps_port</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDS_Octet address[16]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The locators for *Real-Time WAN Transport* use the following mapping:

**Figure 37.2: RTPS WAN Locator**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 37.2 Binding Ping Messages

### Flags

Flags has the following format: x|x|x|x|B|P|U

The B flag indicates whether the locator is unidirectional or bidirectional. Bidirectional locators can send/receive RTPS traffic. Unidirectional locators can only receive RTPS traffic. If the B flag is set, the P flag must be set, too. Locators with the B flag set are called BIDIRECTIONAL locators.

The P flag indicates that the locator contains a public IP address and public port where a DDS endpoint can be reached. **public_ip_address** contains the public IP address, and **public_port** contains the public UDP port. The public UDP port is always used to receive data, and, if the B flag is set, it is also used to send data. Locators with the P flag set are called **PUBLIC locators**.

The U flag indicates whether the locator contains a UUID. While this identifier by itself cannot be directly used to reach a DDS endpoint in a DomainParticipant DP1, the UUID can be mapped to a public address by Cloud Discovery Service and other DomainParticipants. Also, a locator can have both the U flag and the P flag, enabled simultaneously. Locators with the U flag set are called **UUID locators**.

A **PUBLIC+UUID locator** is a locator in which both the U flag and P flag are set.

Initial peers locators will have the B and P flags set and the U flag unset.

The U flag will be set for locators generated automatically by a DomainParticipant.

The P flag will be automatically set for locators generated for a transport that is configured using the property dds.transport.UDPv4_WAN.builtin.public_address. The flag will be also be set by Cloud Discovery Service when generating locators that contain the service reflexive address for a UUID locator.

### 37.2 Binding Ping Messages

As described in 35.1 Setting Real-Time WAN Transport Properties on page 1068, Real-Time WAN Transport uses special RTPS messages called Binding Ping messages to open NAT bindings and to resolve UUID locators into public IP transport addresses.
37.3 Communication Establishment Protocol for Peer-to-Peer Communication with Participants behind Cone NATs

Binding Ping messages contain the UUID and the RTPS port of the locator with which they are associated. This information allows the receiving *Real-Time WAN Transport* to create and update the mapping between a (UUID, RTPS port) pair and its corresponding public address. Figure 37.3: BINARY_PING messages below depicts the structure of a Binding Ping message.

Figure 37.3: BINARY_PING messages

<table>
<thead>
<tr>
<th>0</th>
<th>7</th>
<th>15</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td>BINDING_PING</td>
<td>X X X X B L E</td>
<td>octetsToNextHeader</td>
<td></td>
</tr>
<tr>
<td>DDSUnsignedLong rtps_port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDS_Octet address[12][if L=0]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DDS_Octet address[16][if L=1]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The security of the Binding Pings can be configured using the `com.rti.serv.secure.cryptography.rtps_protection_key` property. For further details, see the "Support for RTI Real-Time WAN Transport" chapter in the *RTI Security Plugins User's Manual*.

**37.3 Communication Establishment Protocol for Peer-to-Peer Communication with Participants behind Cone NATs**

This section describes the communication establishment protocol for the scenario described in **31.2 Peer-to-Peer Communication with Participants behind Cone NATs Using Cloud Discovery Service on page 1048**.

Communication is established as indicated in Figure 37.4: Public Address Resolution Phase Using Cloud Discovery Service (CDS) on the next page and Figure 37.5: UDP Hole Punching Phase on page 1091.
1. **DomainParticipants** DP1 and DP2 register with CDS by sending DDS Participant Announcements PA1 and PA2. Each PA contains two (one for discovery and one for user data) or more UUID locators. These UUID locators are not directly reachable. For the sake of simplicity, **Figure 37.4: Public Address Resolution Phase Using Cloud Discovery Service (CDS)** above only shows the discovery UUID locator being exchanged.

2. When CDS gets the PAs, it obtains the service reflexive address for each one of the UUID locators and updates the PAs, replacing the UUID locators with UUID+PUBLIC locators that contain the service reflexive addresses. UUID+PUBLIC locators are reachable locators.

3. CDS sends PA1', which contains the UUID+PUBLIC locators for DP1, to DP2. It sends PA2', which contains the UUID+PUBLIC locators for DP2, to DP1.

4. After DP1 and DP2 receive each other’s UUID+PUBLIC locators from CDS, they start communicating peer-to-peer using these locators by applying a technique called UDP hole punching.

**Figure 37.5: UDP Hole Punching Phase on the next page** illustrates how UDP hole punching works to allow sending PAs (PA1 and PA2) from DP2 data to DP1. For simplicity, the restricted-cone NAT for DP2 has been removed from the sequence diagram.
In the initial state, DP2 has received a PUBLIC+UUID locator from *Cloud Discovery Service* indicating that DP1 can be reached at the address 40.10.23.45:2000. The PUBLIC+UUID locator was part of PA1' in Figure 37.4: Public Address Resolution Phase Using Cloud Discovery Service (CDS) on the previous page.

1. When DP2 tries to send a PA to DP1, the NAT router for DP1 will drop the message because the NAT binding from 192.168.1.1:100 to 40.10.23.45:2000 does not allow incoming traffic from 50.10.23.445:2000 (see 30.1.1 NAT Kinds on page 1039 for additional details).

2. To allow incoming traffic from DP2, DP1 sends an RTPS BOUNDING_PING message to DP2 public address 50.10.23.445:2000.

3. After the BOUNDING_PING is sent, the NAT router for DP1 will allow PA traffic from DP2 through the NAT binding from 192.168.1.1:100 to 40.10.23.45:2000. For additional details on the BOUNDING_PING message see 37.2 Binding Ping Messages on page 1088.
4. and 5) The next PA announcement coming from DP2 to DP1 will make it through the NAT router for DP1.

The same UDP hole punching mechanism is also used in the opposite direction so that DP1 can send PAs to DP2.

37.4 Communication Establishment Protocol for Peer-to-Peer Communication with a Participant that has a Public Address

This section describes the communication establishment protocol for the scenario described in 31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045.

Communication is established using a technique called “Connection Reversal” as described in Figure 37.6: Connection Reversal Protocol on the next page.
37.4 Communication Establishment Protocol for Peer-to-Peer Communication with a Participant that has a

Figure 37.6: Connection Reversal Protocol

1. DP1 sends a PA to DP2 by using the public IP transport address contained in DP1’s initial peers (50.10.23.45:2000).

2. The PA creates a NAT binding in the DP1 NAT router from 192.168.1.1:1000 to 40.10.23.45:2000 for the destination address 50.10.23.45:2000.

3. The NAT router sends the PA from DP1 to DP2 by replacing the source IP transport address with 40.10.23.45:2000.
4. When DP2 receives the PA from DP1, it will add the discovery UUID locator for DP1 contained in the PA to an unresolved locator table. DP1 cannot send PAs to DP2 yet because it does not know the public IP transport address corresponding to the discovery UUID locator for DP1.

5. DP1 sends a BINDING_PING message from the address associated with the discovery UUID locator to the initial peer for DP2. The initial peer contains the address from which DP2 will send PAs.


7. When DP2 receives the BINDING_PING from DP1, it extracts the source IP transport address (40.10.23.45:2000) from the UDP packet containing the BINDING_PING and associates this address to the unresolved discovery UUID locator from DP1.

8. and 9) At this point, DP2 can send a PA to DP1.
Chapter 38 Transport Debugging

It is recommended that you read Chapter 37 Advanced Concepts on page 1087 before proceeding with this section.

The Real-Time WAN Transport operation can be debugged by setting the Connext DDS verbosity to LOCAL for the COMMUNICATION category:

```xml
<participant_factory_qos>
  <logging>
    <category>COMMUNICATION</category>
    <verbosity>LOCAL</verbosity>
  </logging>
</participant_factory_qos>
```

Or programmatically, shown here in modern C++ (other languages are similar):

```cpp
Logger::instance().verbosity_by_category(LogCategory::COMMUNICATION, Verbosity::STATUS_LOCAL);
```

38.1 Debugging Peer-to-Peer Communication with a Participant that has a Public Address

Consider the communication scenario described in 31.1 Peer-to-Peer Communication with a Participant that has a Public Address on page 1045. Let’s assume the External Participant creates a DataReader on a Topic ‘T’ and the Internal Participant a DataWriter on the same Topic ‘T’.

Figure 38.1: Peer-to-Peer with Any-NAT with a Public Participant

After enabling logging verbosity as indicated above, we can follow the *Real-Time WAN Transport* lifecycle by looking at the logging output:

1. When the Internal Participant is started, it creates a send resource for the locator provided in the initial peers 50.10.23.45:2345:

   \[
   [0x01015FAF,0xBA456FDC,0x040952B8:0x000001C1{D=0}|CREATE\ DP|ENABLE]
   \]

   NDDS_Transport_UDP_create_sendresource_srEA:Created send resource for
   \[f=BP,u\{00,00,00,00,00,00,00,00,00,00,00\},p=50.10.23.45:2345:7410\]

   A send resource is a transport object that can be used to send data to a locator.

   We know this is the initial peer locator because:
   - The locator UUID is all zeros: \(u\{00,00,00,00,00,00,00,00,00,00,00\}\).
   - The public address is the one provided in the initial peers: 50.10.23.45:2345.

   7410 is the RTPS port for discovery data on domain 0. To see how RTPS ports are calculated, see 9.5.9.2 *Ports Used for Discovery* on page 672.

   At this point, the Internal Participant starts sending PAs to the External Participant.

2. When the External Participant receives the PA from the Internal Participant, it creates a send resource for the discovery UUID locator contained in the PA.
38.1 Debugging Peer-to-Peer Communication with a Participant that has a Public Address

The UUID locator is a non-reachable locator.
- \(f=U\) indicates that this is a non-reachable UUID locator.
- \(u\{47,82,BA,ED,A3,27,6F,A8,42\}\) is the UUID of the discovery locator coming from the Internal Participant.
- Even though the public address field has the value \(p=172.31.11.80:0:7410\), the IP address in the log message is not reachable and it corresponds to the private IP address of the Internal Participant.

The External Participant also creates a send resource for the user data UUID locator contained in the PA once it discovers the DataWriter created by the Internal Participant. The user data locator will be used to send RTPS traffic for Topic ‘T’.

The distinction between discovery and user data locators is based on the RTPS port. 7410 is discovery and 7411 is user data.

3. The External Participant cannot send a PA to the Internal Participant until it resolves the public address for the locator with UUID \(\{47,82,BA,ED,A3,27,6F,A8,42\}\) and the RTPS port 7410 associated with the discovery send resource created in step 2.

The public address resolution is done when a BINDING_PING is received from the Internal Participant.
The External Participant also will not be able to send RTPS data for *Topic ‘T’* until it resolves the public address for the locator identified by the UUID \{47,82,BA,ED,A3,27,6F,A8,42\} and the RTPS port 7411. This resolution is also done by the reception of a BINDING_PING:

```
[0x01017AA5,0x4B4691C4,0x600386AD:0x000001C1|PROCESS BINDING PING]
NDSS_Transport_UDPv4_WAN_PublicAddressMappingInfo_log:added
p=7411,u={47,82,BA,ED,A3,27,6F,A8,42},k=3,p=54.151.6.102:7536,f=PU,r=0
```

4. The Internal Participant receives a PA from the External Participant, and it creates two transport send resources: one for sending discovery data and one for sending user data:

**Discovery data:**

```
[0x01010402,0x1CA21E93,0xB02F44C3:0x000100C7\{E=DR,I=21\}|RECEIVE FROM
0x00000000,0x00000000,0x00000000:0x000100C2]
NDSS_Transport_UDP_create_sendresource_srEA:Created send resource for
f=BPU,u={F2,7D,8B,5D,90,AF,93,DD,90},p=50.10.23.45:2345:7410
```

**User data:**

```
[0x01010402,0x1CA21E93,0xB02F44C3:0x80000003\{E=DW,T=Example Quote,C=Quote,D=0\}|LINK
0x01017AA5,0x4B4691C4,0x600386AD:0x80000004\{C=Quote\}]
NDSS_Transport_UDP_create_sendresource_srEA:Created send resource for
f=BPU,u={F2,7D,8B,5D,90,AF,93,DD,90},p=50.10.23.45:2345:7411
```

The user data locator will be used to send RTPS traffic for *Topic ‘T’*. Note that the public address 50.10.23.45:2345 is the same for both send resources because we are configuring the External Participant to use a single UDP port for communications. The distinction between user data and discovery data is done by looking at the RTPS port. 7410 is the port for discovery and 7411 is the port for user data.

5. At this point, both Participants can communicate with each other. The External Participant will start receiving samples for *Topic ‘T’* from the Internal Participant.

### 38.2 Peer-to-Peer Communication with Participants behind Cone NATs

This section covers the scenario described in **31.2 Peer-to-Peer Communication with Participants behind Cone NATs Using Cloud Discovery Service on page 1048**. It is recommended that you read that section to interpret some of the log messages.
38.2 Peer-to-Peer Communication with Participants behind Cone NATs

Figure 38.2: Peer-to-Peer with Cone NATs

The Internal Participant 1 (DP1) will create a DataWriter publishing data on a Topic ‘T’, and the Internal Participant 2 (DP2) will create a DataReader subscribing to Topic ‘T’.

This section will focus on debugging the Real-Time WAN Transport lifecycle for DP1 and DP2. For details on how to debug Cloud Discovery Service (CDS), see "Debugging Cloud Discovery Service with the UDP WAN Transport," in the NAT Traversal section of the RTI Cloud Discovery Service documentation.

1. When DP1 is started, it creates a send resource for the locator provided as the initial peer 50.10.23.45:2345. This locator corresponds to the CDS locator.

```
[0x01016F1B,0x2294D448,0x8060E06B:0x000001C1{D=0}|CREATE DP|ENABLE]
NDDS_Transport_UDP_create_sendresource_srEA:Created send resource for
f=BP,u={00,00,00,00,00,00,00,00},p=50.10.23.45:2345:2345
```

We know this is the initial peer locator because:
- The locator UUID is all zeros: u={00,00,00,00,00,00,00,00}.
- The public address is the one provided in the initial peers: 50.10.23.45:2345.

At this point the DP1 starts sending PAs to CDS.
2. Likewise, when DP2 is started, it creates a send resource for the CDS locator provided as the initial peer 50.10.23.45:2345.

```
[0x01017116,0xF14A169C,0xE7799A94:0x000001C1|CREATE_DP|ENABLE]
NDDS_Transport_UDP_create_sendresource_srEA:Created send resource for
f=BP,u={00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00,00},p=50.10.23.45:2345:2345
```

At this point, DP2 starts sending PAs to CDS.

3. To start sending PAs to DP2, DP1 must receive a PA from CDS on behalf of DP2 containing the discovery UUID+PUBLIC locator at which DP2 can be reached.

4. Once DP1 receives the PA from CDS, it creates a send resource for the discovery UUID+PUBLIC locator used for discovery and starts sending PAs to DP2.

```
[0x01016F1B,0x2294D448,0x8060E06B:0x000100C7|RECEIVE_FROM
0x00000000,0x00000000,0x00000000:0x000100C2]
NDDS_Transport_UDP_create_sendresource_srEA:Created send resource for
f=BPU,u={50,26,6D,EA,B7,11,AC,B9,5F},p=99.35.17.233:7535:7410
```

99.35.17.233:7535 is the public address at which DP2 will receive RTPS discovery traffic.

5. To start sending PAs to DP1, DP2 must receive a PA from CDS on behalf of DP1 containing the discovery UUID+PUBLIC locator at which DP1 can be reached.

6. Once DP2 receives the PA from CDS, it creates a send resource for the UUID+PUBLIC locator used for discovery and starts sending PAs to DP1.

```
[0x01017116,0xF14A169C,0xE7799A94:0x000100C7|RECEIVE_FROM
0x00000000,0x00000000,0x00000000:0x000100C2]
NDDS_Transport_UDP_create_sendresource_srEA:Created send resource for
f=BPU,u={B1,1D,4B,B5,A1,58,5E,E1,58},p=54.151.6.102:7535:7410
```

54.151.6.102:7535 is the public address at which DP 1 will receive RTPS discovery traffic.

7. After DP1 discovers DP2’s DataReader for Topic ‘T’, DP1 will create a send resource to send RTPS data for Topic ‘T’ (samples, GAPs, and HBs) to the DataReader in DP2.

```
[0x01016F1B,0x2294D448,0x8060E06B:0x80000003|E=DW,T=Example Quote,C=Quote,D=0]|LINK
0x01017116,0xF14A169C,0xE7799A94:0x80000004|C=Quote]
NDDS_Transport_UDP_create_sendresource_srEA:Created send resource for
f=PU,u={50,26,6D,EA,B7,11,AC,B9,5F},p=99.35.17.233:7536:7411
```

99.35.17.233:7536 is the public address at which DP1 will receive RTPS user data traffic for Topic ‘T’.

8. After DP2 discovers DP1’s DataWriter for Topic ‘T’, DP2 will create a send resource to send RTPS data for Topic ‘T’ (NACKs) to the DataWriter in DP1.

```
[0x01017116,0xF14A169C,0xE7799A94:0x80000004|E=DR,T=Example Quote,C=Quote,D=0]|LINK
0x01016F1B,0x2294D448,0x8060E06B:0x80000003|C=Quote]
NDDS_Transport_UDP_create_sendresource_srEA:Created send resource for
```
38.2 Peer-to-Peer Communication with Participants behind Cone NATs

Note that in this case, the send resource has been created with a UUID locator that is not reachable. When this occurs, the public IP transport address for the UUID locator will be resolved by receiving a BINDING_PING from DP1.

9. At this point, both Participants can communicate with each other. DP2's DataReader will start receiving samples for Topic ‘T’ from DP1’s DataWriter.
Chapter 39 Tools Integration

RTI Tools such as RTI Admin Console can use Real-Time WAN Transport if they are configured appropriately.

Admin Console ships with a built-in profile that enables use of the Real-Time WAN Transport: AdminConsole::RealTimeWAN. Make sure you select that profile in the Admin Console Preferences and provide the right initial peers (see Chapter 34 Transport Initial Peers on page 1066) to Admin Console to inspect Connext DDS applications running across the WAN. See Figure 39.1: Real-Time WAN Transport and Admin Console on the next page.

```xml
<qos_library name="AdminConsole">
  <qos_profile name="RealTimeWAN" base_name="AdminConsole::Default">
    <participant_qos>
      <transport_builtin>
        <mask>MASK_DEFAULT|UDPv4_WAN</mask>
      </transport_builtin>
    </participant_qos>
  </qos_profile>
</qos_library>
```

**Important:** The auto-join feature will not work when using Real-Time WAN Transport because multicast is not available in WAN environments. You will have to join the WAN domain(s) explicitly.
Figure 39.1: *Real-Time WAN Transport and Admin Console*

Select `AdminConsole::RealTimeWAN`

Enter initial peers

Manually join/leave
Chapter 40 Troubleshooting

40.1 Communication Stops Working after Application Transitions to Different Network

Possible Root Cause:

If you are using Cloud Discovery Service, it is possible that before the network transition, all the applications were behind cone NATs. When the transition occurs, the application that is changing to a new network connects to a symmetric NAT.

This configuration is not currently supported. Symmetric NATs are only supported in scenarios like the one described in Figure 31.1: Peer-to-Peer between a Participant behind Any Kind of NAT and an External Participant on page 1046.

To validate if you are behind a symmetric NAT in the new network, you can run the application natat as described in 30.1.2 Identifying the NAT Type on page 1041.

Solution:

When you do not know in advance whether the networks in which applications run are behind cone NATs or symmetric NATs, you may want to do a relay deployment as indicated in 32.2 Relayed Edge-to-Edge Deployment Scenario on page 1057.

40.2 Communication not Established after Changing Cloud Discovery Service <receiver_port>

For example, when changing the Cloud Discovery Service configuration from:

```xml
<dds>
  <cloud_discovery_service name="CDS">
    <transport>
      <element>
        <alias>builtin.udpv4_wan</alias>
        <receive_port>2345</receive_port>
        <property>
```
To:

Possible Root Cause:

If Cloud Discovery Service (CDS) is running behind a NAT-enabled router, it is possible that you have not created a static NAT binding in the router for the new public address: 50.10.23.45:6001.

Solution:

Create a new static NAT binding to support the port change. The NAT binding must create this mapping:

\[
\text{<CDS private address X>:<port number Y> → <new public address Z>:<port number Y>}
\]

\[
\text{<CDS private address>:6001 → 50.10.23.45:6001}
\]

Note that the private host CDS port and public port must be the same (6001 is the port number for both in the example above). To make them different, use the transport property \texttt{dds.transport.UDPv4\_WAN.builtin.comm\_ports}. See 35.1 Setting Real-Time WAN Transport Properties on page 1068.

40.3 Communication not Established even though Transport Settings are Set Correctly

Possible Root Cause:
There may be an IP fragmentation problem. For WAN communications, it is not a good idea to rely on IP fragmentation. IP fragmentation causes significant issues in UDP, where there is no support for an MTU (maximum transmission unit) discovery protocol as there is in TCP. These are some of the problems associated with IP fragmentation:

- To successfully reassemble a packet, all fragments must be delivered. No fragment can become corrupt or get lost in-flight. If a fragment is lost, the whole packet will be lost.
- Before reassembly, a host must hold partial, fragment datagrams in memory. This opens an opportunity for memory exhaustion attacks.
- Subsequent fragments lack the higher-layer header. The TCP or UDP header is only present in the first fragment, making it impossible for firewalls to filter fragment datagrams based on criteria like source or destination ports.

When testing over some cellular networks, in some cases you may not be able to send samples larger than the IP MTU, such as images, without losing a large percentage of the frames.

**Solution:**

Disable IP fragmentation by letting Connext DDS do fragmentation at the RTPS level. For details, see section 35.3 Disabling IP Fragmentation for Real-Time WAN Transport on page 1084.

### 40.4 "serialize buffer too small" Error

**Possible Root Cause:**

Disabling IP fragmentation and reducing the protocol overhead as described in 35.3 Disabling IP Fragmentation for Real-Time WAN Transport on page 1084 will result in a better payload utilization ratio, but comes with the risk that there is not enough room left to store the required meta-information. If the `rtps_overhead` property value is too small, you will see the following error:

```
MIGGenerator_addDataFragEx:serialize buffer too small
COMMENDSrWriterService_agentFunction:!add DataFrag
```

**Solution:**

You can increase the `rtps_overhead` property to a value that fits your use case by using this table:

<table>
<thead>
<tr>
<th>RTPS Message</th>
<th>Overhead (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular sample</td>
<td>92</td>
</tr>
<tr>
<td>Batch sample</td>
<td>98</td>
</tr>
<tr>
<td>TopicQuery/Multichannel sample</td>
<td>96</td>
</tr>
</tbody>
</table>
40.5 Slow Discovery using Cloud Discovery Service

<table>
<thead>
<tr>
<th>Sample Inline QoS (per sample)</th>
<th>Overhead (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyed Topic</td>
<td>28</td>
</tr>
<tr>
<td>Content filter (one DataReader per Topic per DomainParticipant)</td>
<td>32</td>
</tr>
<tr>
<td>Content filter (additional DataReader per Topic per DomainParticipant up to 3)</td>
<td>16</td>
</tr>
<tr>
<td>Zero Copy transfer over shared memory</td>
<td>8</td>
</tr>
<tr>
<td>DataWriter coherent set</td>
<td>12</td>
</tr>
<tr>
<td>Group coherent set</td>
<td>32</td>
</tr>
<tr>
<td>TopicQuery</td>
<td>40</td>
</tr>
<tr>
<td>Request-Reply</td>
<td>28</td>
</tr>
<tr>
<td>Batching: Header</td>
<td>8</td>
</tr>
<tr>
<td>Batching: Timestamp</td>
<td>8</td>
</tr>
<tr>
<td>Batching: Distance</td>
<td>4</td>
</tr>
<tr>
<td>RTI Persistence Service or RTI Routing Service with &lt;publish_with_original_info&gt; set to TRUE</td>
<td>28</td>
</tr>
<tr>
<td>RTI Queuing Service</td>
<td>96</td>
</tr>
</tbody>
</table>

The recommended number of 196 for the **rtps_overhead** in section 35.3 Disabling IP Fragmentation for Real-Time WAN Transport on page 1084 comes from considering the following configuration: (Regular sample + Keyed Topic + Content filter + Routing Service overhead = 180) + (safety margin = 16).

For example, if you use DataWriter coherent sets, add 12 bytes to the recommended value.

To use the table, pick a baseline: Regular sample, Batch sample, or TopicQuery/Multichannel sample. Then add the overhead of specific features that you use in your system (e.g., content filter) to that baseline.

**40.5 Slow Discovery using Cloud Discovery Service**

**Possible Root Cause:**

If you are using *Cloud Discovery Service*, it is possible that the Participant Announcements sent by *Cloud Discovery Service* to other DomainParticipants when discovering a new DomainParticipant are lost. This may delay the discovery process.

**Solution:**

To accelerate the discovery process, decrease the refresh period from the default of 60 seconds to a smaller value (e.g., 10 seconds). For example:

```xml
<dds>
  <cloud_discovery_service name="CDS">
    <transport>
```
40.5 Slow Discovery using Cloud Discovery Service

```xml
<element>
  <alias>builtin.udpv4_wan</alias>
  <receive_port>2345</receive_port>
  <property>
    <element>
      <name>dds.transport.UDPv4_WAN.builtin.public_address</name>
      <value>60.10.23.45</value>
    </element>
  </property>
</element>
</transport>
<forwarder>
  <event>
    <refresh_period>
      <sec>10</sec>
      <nanosec>0</nanosec>
    </refresh_period>
  </event>
</forwarder>
</cloud_discovery_service>
</dds>
```
Part 6: RTI Secure WAN Transport

Secure WAN Transport is an optional package that enables participant discovery and data exchange in a secure manner over the public WAN. Secure WAN Transport enables Connext DDS to address the challenges in NAT traversal and authentication of all participants. By implementing UDP hole punching using the STUN protocol and providing security to channels by leveraging DTLS (Datagram TLS), you can securely exchange information between different sites separated by firewalls.

The material in this part of the manual is only relevant if you have installed Secure WAN Transport. This feature is not installed as part of a Connext DDS package; it must be downloaded and installed separately. It is only available on specific architectures. See the RTI Secure WAN Transport Release Notes and RTI Secure WAN Transport Installation Guide for details.

RTI may not support Secure WAN Transport in future versions of Connext DDS. Existing applications that use it should be updated to take advantage of Real-Time WAN Transport as soon as feasible. All new applications should use Real-Time WAN Transport. See Part 5: RTI Real-Time WAN Transport on page 1034.

This section includes:

- Introduction to Secure WAN Transport (Chapter 41 on page 1110)
- Configuring RTI Secure WAN Transport (Chapter 42 on page 1122)
Secure WAN Transport provides transport plugins that can be used by developers of Connext DDS applications. These transport plugins allow Connext DDS applications running on private networks to communicate securely over a Wide-Area Network (WAN), such as the internet. There are two primary components in the package which may be used independently or together: communication over Wide-Area Networks that involve Network Address Translators (NATs), and secure communication with support for peer authentication and encrypted data transport.

The Connext DDS core is transport-agnostic. Connext DDS offers three built-in transports: UDP/IPv4, UDP/IPv6, and inter-process shared memory. The implementation of NAT traversal and secure communication is done at the transport level so that the Connext DDS core is not affected and does not need to be changed, although there is additional on-the-wire traffic.

The basic problem to overcome in a WAN environment is that messages sent from an application on a private local-area network (LAN) appear to come from the LAN’s router address, not from the internal IP address of the host running the application. This is due to the existence of a Network Address Translator (NAT) at the gateway. This does not cause problems for client/server systems because only the server needs to be globally addressable; it is only a problem for systems with peer-to-peer communication models, such as Connext DDS. Secure WAN Transport solves this problem, allowing communication between peers that are in separate LAN networks, using a UDP hole-punching mechanism based on the STUN protocol (IETF RFC 3489bis) for NAT traversal. This requires the use of an additional rendezvous server application, the RTI WAN Server.

Once the transport has enabled traffic to cross the NAT gateway to the WAN, it is flowing on network hardware that is shared (in some cases, over the public internet). In this context, it is important to consider the security of data transmission. There are three primary issues involved:

- Authenticating the communication peer (source or destination) as a trusted partner;
- Encrypting the data to hide it from other parties that may have access to the network;
Validating the received data to ensure that it was not modified in transmission.

Secure WAN Transport addresses these problems by wrapping all RTPS-encoded data using the DTLS protocol (IETF RFC 4347), which is a variant of SSL/TLS that can be used over a datagram network-layer transport such as UDP. The security features of the WAN Transport may also be used on an untrusted local-area network with the Secure Transport.

In summary, the package includes two transports:

- The WAN Transport is for use on a WAN and includes security. It must be used with the WAN Server, a rendezvous server that provides the ability to discover public addresses and to register and look up peer addresses based on a unique WAN ID. The WAN Server is based on the STUN (Session Traversal Utilities for NAT) protocol [https://tools.ietf.org/html/rfc5389], with some extensions. Once information about public addresses for the application and its peers has been obtained and connections have been initiated, the server is no longer required to maintain communication with a peer. (Note: security is disabled by default.)
- The Secure Transport is an alternate transport that provides security on an untrusted LAN. Use of the RTI WAN Server is not required.

Multicast communication is not supported by either of these transports.

This chapter provides a technical overview of:

- 41.1 WAN Traversal via UDP Hole-Punching below
- 41.2 WAN Locators on page 1115
- 41.3 Datagram Transport-Layer Security (DTLS) on page 1116
- 41.4 Certificate Support on page 1118

For information on how to use Secure WAN Transport with your Connext DDS application, see Configuring RTI Secure WAN Transport (Chapter 42 on page 1122).

41.1 WAN Traversal via UDP Hole-Punching

In order to resolve the problem of communication across NAT boundaries, the WAN Transport implements a UDP hole-punching solution for NAT traversal [draft-ietf-behave-p2p-state]. This solution uses a rendezvous server, which provides the ability to discover public addresses, and to register and lookup peer addresses based on a unique WAN ID. This server is based on the STUN (Session Traversal Utilities for NAT) protocol [draft-ietf-behave-rfc3489bis], with some extensions. This protocol is a part of the solution used for standards-based voice over IP applications; similar technology has been used by systems such as Skype and has proven to be highly reliable. A key advantage of STUN is that it is based on UDP and therefore is able to preserve the real-time characteristics of the DDS Interoperability Wire Protocol.
Once information about public addresses for the application and its peers has been obtained, and connections have been initiated, the server is no longer required to maintain communication with a peer. However, if communication fails, possibly due to changes in dynamically-allocated addresses, the server will be needed to reopen new public channels.

Figure 41.1: RTI WAN Transport Architecture below shows the RTI WAN transport architecture.

**Figure 41.1: RTI WAN Transport Architecture**

---

### 41.1.1 Protocol Details

The UDP hole-punching algorithm implemented by the WAN transport has two different phases: registration and connection. This algorithm only works with cone or asymmetric NATs where the same public address/port is assigned to all the sessions with the same private address/port address.

- **Registration Phase**

  The RTI WAN Server application runs on a machine that resides on the WAN network (i.e., not in a private LAN). It has to be globally accessible to LAN applications. It is started by a script and acts as a rendezvous point for LAN applications. During the registration phase, each transport locator is registered with the RTI WAN Server using a STUN binding request message.

  The RTI WAN Server associates RTPS locators with their corresponding public IPv4 transport addresses (a combination of IP address and port) and stores that information in an internal table. **Figure 41.2: Registration Phase on the next page** illustrates the registration phase.
Figure 41.2: Registration Phase

- **Connection Phase**

The connection phase starts when locator A wants to establish a connection with locator B. Locator A obtains information about locator B via Connext DDS discovery traffic or the initial NDDS_DISCOVERY_PEERS list. To establish a connection with locator B, locator A sends a STUN connect request to the RTI WAN server. The server sends a STUN connect response to locator A, including information about the public IP transport address (IP address and port) of locator B. In parallel, the RTI WAN server contacts locator B using another STUN connect request to let it know that locator A wants to establish a connection with it.

When locator A receives the public IP address of locator B, it will try to contact B using two STUN binding request messages. The first message is sent to the public address of B and the second message is sent to the private address of B. The private address was obtained using the last 32 bits of the locator address of B. The STUN binding request message directed to the public transport address of B sent by locator A will open a hole in A’s NAT to receive messages from B.
When locator B receives the public address of locator A, it will try to contact A sending a STUN binding request message to that public address. This message will open a hole in B’s NAT to receive messages from A. When locator A receives the first STUN binding response from locator B, it starts sending RTPS traffic.

The connection phase includes two processes: the connect process (Figure 41.3: Connect Process below) and the NAT hole punching process (Figure 41.4: NAT Hole Punching Process on the next page).

Figure 41.3: Connect Process
Finally, since bindings allocated by NAT expire unless refreshed, the clients (locators) must generate binding request messages for the server and other clients to refresh the bindings. The RTI STUN protocol implementation uses the attribute LIVELINESS-PERIOD in the STUN binding request to indicate the period in milliseconds at which a client will assert its liveliness. The WAN Server will remove a locator from its mapping table when the liveliness contract is not met. Likewise, a transport instance will remove a STUN connection with a locator when this locator does not assert its liveliness as indicated in the last binding request.

41.2 WAN Locators

The WAN transport does not use simple IP addresses to locate peers. A WAN transport locator consists of a WAN ID, which is an arbitrary 12-byte value, and a bottom 4-byte value that specifies a fallback local IPv4 address. Your peers list (NDDS_DISCOVERY_PEERS) must be configured to look for peers with locators of the form:
41.3 Datagram Transport-Layer Security (DTLS)

Data security is provided by wrapping all Connext DDS network traffic with the Datagram Transport Layer Security (DTLS) protocol (IETF RFC 4347). DTLS is a relatively recent variant of the mature SSL/TLS family of protocols which adds the capability to secure communication over a connectionless network-layer transport such as UDP. UDP is the preferred network layer transport for the DDS wire protocol RTPS, as well as for NAT traversal. Like SSL/TLS, the DTLS protocol provides capabilities for certificate-based authentication, data encryption, and message integrity. The protocol specifies a number of standard cryptographic algorithms that must be available; the base set is listed in the TLS 1.1 specification (IETF RFC 4346).

Secure protocol support is provided by the open source OpenSSL library, which has supported the DTLS protocol since the release of OpenSSL 0.9.8. Note however that many critical issues in DTLS were resolved by the OpenSSL 0.9.8f release. For more detailed information about available ciphers, certificate support, etc. please refer to the OpenSSL documentation. The DTLS protocol securely authenticates with each individual peer; as such, multicast communication is not supported by the Secure Transport. There is also a FIPS security-certified version of OpenSSL (OpenSSL-FIPS 1.1.1), but this does not yet support DTLS.

The Secure Transport protocol stack is similar to the Secure WAN transport stack, but without the STUN layer and server. See Figure 41.5: DTLS Architecture on the next page.

- The address is a 128-bit address in IPv6 notation.
- The first part, "3@", specifies the maximum participant ID to be contacted at the given locator. See 15.2.1 Peer Descriptor Format on page 769.
- The "wan://" part specifies that the address is for the WAN transport.
- The next part, ":::1", specifies the top 12 bytes of the address to be 11 zero bytes, followed by a byte with value 1 (this corresponds to the peer's WAN ID). The WAN ID must be in hexadecimal format. For example, if the WAN ID was decimal 23, the locator would be:
  3@wan://::17:10.10.1.150 (not 3@wan://::23:10.1.150).
- The last part, "10.10.1.150" refers to the peers local IPv4 address, which will be used if the peers are on the same local network.

A DomainParticipant using the WAN transport will have to initialize the DDS_DiscoveryQosPolicy’s initial_peers field with the WAN locator addresses corresponding to the peers to which it wants to connect to. The value of initial_peers can be set using the environment variable NDDS_DISCOVERY_PEERS or the NDDS_DISCOVERY_PEERS configuration file. (See 15.2 Configuring the Peers List Used in Discovery on page 767.)
41.3.1 Security Model

In order to communicate securely, an instance of the secure plugin requires: 1) a certificate authority (shared with all peers), 2) an identifying certificate which has been signed by the authority, 3) the private key associated with the public key contained in the certificate.

The Certificate Authority (CA) is specified by using a PEM format file containing its public key or by using a directory of PEM files following standard OpenSSL naming conventions. If a single CA file is used, it may contain multiple CA keys. In order to successfully communicate with a peer, the CA keys that are supplied must include the CA that has signed that peer's identifying certificate.

The identifying certificate is specified by using a PEM format file containing the chain of CAs used to authenticate the certificate. The identifying certificate must be signed by a CA. It will either be directly signed by a root CA (one of the CAs supplied above), by an authority whose certificate has been signed by the root CA, or by a longer chain of certificate authorities. The file must be sorted starting with the certificate to the highest level (root CA). If the certificate is directly signed by a root CA, then this file will only contain the root CA certificate followed by the identity certificate.

Finally, a private key is required. In order to avoid impersonation of an identity, this should be kept private. It can be stored in its own PEM file specified in one of the private key properties, or it can be appended to the certificate chain file.

One complication in the use of DTLS for communication by Connext DDS is that even though DTLS is a connectionless protocol, it still has client/server semantics. The RTI Secure Transport maps a bidirectional

Figure 41.5: DTLS Architecture
communication channel between two peer applications into a pair of unidirectional encrypted channels. Both peers are playing the part of a client (when sending data) and a server (when receiving).

### 41.3.2 Liveliness Mechanism

When a peer shuts down cleanly, the DTLS protocol ensures that resources are released. If a peer crashes or otherwise stops responding, a liveliness mechanism in the DTLS transport cleans up resources. You can configure the DTLS handshake retransmission interval and the connection liveliness interval.

### 41.4 Certificate Support

Cryptographic certificates are required to use the security features of the WAN transport. This section describes a mechanism to use the OpenSSL command line tool to generate a simple private certificate authority. For more information, see the manual page for the `openssl` tool ([http://www.openssl.org/docs/apps/openssl.html](http://www.openssl.org/docs/apps/openssl.html)) or the book, "Network Security with OpenSSL" by Viega, Messier, & Chandra (O'Reilly 2002), or other references on Public Key Infrastructure.

1. Initialize the Certificate Authority:
   a. Create a copy of the `openssl.cnf` file and edit fields to specify the proper default names and paths.
   b. Create the required CA directory structure:

   ```
   mkdir myCA
   mkdir myCA/certs
   mkdir myCA/private
   mkdir myCA/newcerts
   mkdir myCA/crl
   touch myCA/index.txt
   ```
   c. Create a self-signed certificate and CA private key:

   ```
   openssl req -nodes -x509 -days 1095 -newkey rsa:2048 \
   -keyout myCA/private/cakey.pem -out myCA/cacert.pem \
   -config openssl.cnf
   ```

2. For each identifying certificate:
   a. You may want to create a copy of your customized `openssl.cnf` file with default identifying information to be used as a template for certificate request creation; the commands below refer to this file as `template.cnf`.
   b. Generate a certificate request and private key:

   ```
   openssl req -nodes -new -newkey rsa:2048 -config template.cnf \
   -keyout peer1key.pem -out peer1req.pem
   ```
   c. Use the CA to sign the certificate request to generate certificate:
41.5 License Issues

The OpenSSL toolkit stays under a dual license, i.e., both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

```c
/* Copyright (c) 1998-2007 The OpenSSL Project. All rights reserved. */
/* */
/* Redistribution and use in source and binary forms, with or without */
/* modification, are permitted provided that the following conditions */
/* are met: */
/* */
/* 1. Redistributions of source code must retain the above copyright */
/* notice, this list of conditions and the following disclaimer. */
/* 2. Redistributions in binary form must reproduce the above copyright */
/* notice, this list of conditions and the following disclaimer in */
/* the documentation and/or other materials provided with the */
/* distribution. */
/* 3. All advertising materials mentioning features or use of this */
/* software must display the following acknowledgment: */
/* "This product includes software developed by the OpenSSL Project */
/* for use in the OpenSSL Toolkit. (http://www.openssl.org/)" */
/* 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to */
/* endorse or promote products derived from this software without */
/* prior written permission. For written permission, please contact */
/* openssl-core@openssl.org. */
/* */
/* 5. Products derived from this software may not be called "OpenSSL" */
/* nor may "OpenSSL" appear in their names without prior written */
/* permission of the OpenSSL Project. */
/* */
/* 6. Redistributions of any form whatsoever must retain the following */
/* acknowledgment: */
/* "This product includes software developed by the OpenSSL Project */
/* for use in the OpenSSL Toolkit (http://www.openssl.org/)" */
/* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ''AS IS'' AND ANY */
/* EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE */
/* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR */
/* PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE OpenSSL PROJECT OR */
/* ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, */
/* SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT */
/* NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; */
/* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) */
/* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, */
/* STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) */
```

d. Optionally, append the private key to the peer certificate:

```bash
cat myCA/newcerts/peer1cert.pem peer1key.pem |
> ${private location}/peer1.pem
```
* ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
* OF THE POSSIBILITY OF SUCH DAMAGE.
*=====================================================================
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com). This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*/

Original SSLeay License
-----------------------
/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
* All rights reserved.
* This package is an SSL implementation written
* by Eric Young (eay@cryptsoft.com).
* The implementation was written so as to conform with Netscapes SSL.
* This library is free for commercial and non-commercial use as long as
* the following conditions are heared to. The following conditions
* apply to all code found in this distribution, be it the RC4, RSA,
* lhash, DES, etc., code; not just the SSL code. The SSL documentation
* included with this distribution is covered by the same copyright terms
* except that the holder is Tim Hudson (tjh@cryptsoft.com).
* Copyright remains Eric Young's, and as such any Copyright notices in
* the code are not to be removed.
* If this package is used in a product, Eric Young should be given
* attribution
* as the author of the parts of the library used.
* This can be in the form of a textual message at program startup or
* in documentation (online or textual) provided with the package.
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions
* are met:
* 1. Redistributions of source code must retain the copyright
*    notice, this list of conditions and the following disclaimer.
* 2. Redistributions in binary form must reproduce the above copyright
*    notice, this list of conditions and the following disclaimer in the
*    documentation and/or other materials provided with the distribution.
* 3. All advertising materials mentioning features or use of this software
*    must display the following acknowledgement:
*       "This product includes cryptographic software written by
*       Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the routines from the
*    library
*    being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof)
*    from the apps directory (application code) you must include an
*    acknowledgement:
*       "This product includes software written by Tim Hudson
*        (tjh@cryptsoft.com)"
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
41.5 License Issues

* PURPOSE
* ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publicly available
* version or
* derivative of this code cannot be changed. i.e. this code cannot
* simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.] */
Chapter 42 Configuring RTI Secure WAN Transport

The Secure WAN Transport package includes two transports:

- **The WAN Transport** is for use on a WAN and includes security.\(^1\) It must be used with the WAN Server, a separate application that provides additional services needed for Connext DDS applications to communicate with each other over a WAN.

- **The Secure Transport** is an alternate transport that provides security on an untrusted LAN. Use of the RTI WAN Server is not required.

There are two ways in which these transports can be configured:

- By setting up predefined strings in the Property QoS Policy of the DomainParticipant, on architectures that support dynamic libraries. This process is described in 42.2 Setting Up a Transport with the Property QoS on the next page.

  Dynamic libraries are supported on all architectures except INTEGRITY and certain VxWorks architectures. For VxWorks, dynamic libraries are only supported for architectures that are on Pentium®/Arm® CPUs AND use kernel mode.

- By instantiating a new transport (42.5 Explicitly Instantiating a WAN or Secure Transport Plugin on page 1139) and then registering it with the DomainParticipant, see 16.7 Installing Additional Builtin Transport Plugins with register_transport() on page 821 (not available in the Java API).

Refer to the API Reference HTML documentation for details on these two approaches.

---

\(^1\)Security is disabled by default.
### 42.1 Example Applications

A simple example is available to show how to configure the WAN transport. It includes example settings to enable communication over WAN, and optional settings to enable security (along with example certificate files to use for secure communication). The example is located in `<path to examples>/connext_dds/<language>/hello_world_wan`.

As seen in the example, you can configure the properties of either transport by setting the appropriate name/value pairs in the `DomainParticipant`'s PropertyQoS, as described in 42.2 Setting Up a Transport with the Property QoS below. This will cause Connext DDS to dynamically load the WAN or Secure Transport libraries at run time and then implicitly create and register the transport plugin.

Another way to use the WAN or Secure transports is to explicitly create the plugin and use `register_transport()` to register the transport with *Connext DDS* (see 16.7 Installing Additional Builtin Transport Plugins with `register_transport()` on page 821). This way is *not* shown in the example. See 42.5 Explicitly Instantiating a WAN or Secure Transport Plugin on page 1139.

### 42.2 Setting Up a Transport with the Property QoS

The 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 allows you to set up name/value pairs of data and attach them to an entity, such as a `DomainParticipant`. This will cause Connext DDS to dynamically load the WAN or Secure Transport libraries at run time and then implicitly create and register the transport plugin.

Please refer to 16.6 Setting Builtin Transport Properties with the PropertyQosPolicy on page 801.

To assign properties, use the `add_property()` operation:

```cpp
DDS_ReturnCode_t DDSPROPERTYQOSPolicyHelper::add_property
    (DDSPROPERTYQOSPolicy policy,
     const char * name,
     const char * value,
     DDS_BOOLEAN propagate)
```

For more information on `add_property()` and the other operations in the DDSPROPERTYQOSPolicyHelper class, please see Table 7.65 PropertyQoSPolicyHelper Operations, as well as the API Reference HTML documentation.

The ‘name’ part of the name/value pairs is a predefined string, described in 42.3 WAN Transport Properties on page 1125 and 42.4 Secure Transport Properties on page 1133.

Here are the basic steps, taken from the example Hello World application (for details, please see the example application.)

---

\(^a\)See Paths Mentioned in Documentation on page 1.
1. Get the default DomainParticipant QoS from the DomainParticipantFactory.

```
DDSDomainParticipantFactory::get_instance()->
  get_default_participant_qos(participant_qos);
```

2. Disable the builtin transports.

```
participant_qos.transport_builtin.mask =
  DDS_TRANSPORTBUILTIN_MASK_NONE;
```

3. Set up the DomainParticipant's Property QoS.
   a. Load the plugin.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property,
  "dds.transport.load_plugins",
  "dds.transport.wan_plugin.wan",
  DDS_BOOLEAN_FALSE);
```

   b. Specify the transport plugin library.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property,
  "dds.transport.wan_plugin.wan.library",
  "libnddstransportwan.so",
  DDS_BOOLEAN_FALSE);
```

   c. Specify the transport’s ‘create’ function.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property,
  "dds.transport.wan_plugin.wan.create_function"
  "NDDS_Transport_WAN_create",
  DDS_BOOLEAN_FALSE);
```

   d. Specify the WAN Server and instance ID.

```
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property
  "dds.transport.wan_plugin.wan.server",
  "192.168.1.1",
  DDS_BOOLEAN_FALSE);
DDSPropertyQosPolicyHelper::add_property (
  participant_qos.property
  "dds.transport.wan_plugin.wan.transport_instance_id",
  1,
  DDS_BOOLEAN_FALSE);
```

   e. Specify any other properties, as needed.

4. Create the DomainParticipant, using the modified QoS.
```
participant = DDSTheParticipantFactory->create_participant(
    domainId,
    participant_qos,
    NULL /* listener */,
    DDS_STATUS_MASK_NONE);
```

Property changes should be made before the transport is loaded: either before the `DomainParticipant` is enabled, before the first `DataWriter/DataReader` is created, or before the builtin topic reader is looked up, whichever one happens first.

### 42.3 WAN Transport Properties

Table 42.1 Properties for NDDS_Transport_WAN_Property_t lists the properties that you can set for the WAN Transport.

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dds.transport.load_plugins</code></td>
<td>Required</td>
</tr>
<tr>
<td><em>(note: this does not take a prefix)</em></td>
<td>Comma-separated strings indicating the prefix names of all plugins that will be loaded by Connext DDS. You will use this string as the prefix to the property names. For example: &quot;dds.transport.WAN.wan1&quot;. <em>(This assumes you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins.)</em> This prefix must begin with 'dds.transport.' Note: You can load up to 8 plugins.</td>
</tr>
<tr>
<td><code>library</code></td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Must set to nddstransportwan. This library and the dependent OpenSSL libraries need to be in your library search path (pointed to by the environment variable LD_LIBRARY_PATH on Linux systems, Path on Windows systems, DYLD_LIBRARY_PATH on macOS systems).</td>
</tr>
<tr>
<td><code>create_function</code></td>
<td>Required</td>
</tr>
<tr>
<td></td>
<td>Must be &quot;NDDS_Transport_WAN_create&quot;.</td>
</tr>
</tbody>
</table>

1 Assuming you used ‘dds.transport.WAN.wan1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
Table 42.1 Properties for NDDS_Transport_WAN_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.WAN.wan1.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aliases</td>
<td>Used to register the transport plugin returned by NDDS_Transport_WAN_create() (as specified by &lt;WAN_prefix&gt;.create_function) to the DomainParticipant. Aliases should be specified as a comma-separated string, with each comma delimiting an alias. If it is not specified, the prefix—without the leading &quot;dds.transport&quot;—is used as the default alias for the plugin. For example, if the &lt;WAN_prefix&gt; is &quot;dds.transport.mytransport&quot;, the default alias for the plugin is &quot;mytransport&quot;.</td>
</tr>
</tbody>
</table>
| verbosity                                               | Specifies the verbosity of log messages from the transport. Possible values:  
  - -1: silent  
  - 0 (default): errors only  
  - 1: errors and warnings  
  - 2: local status  
  - 5 or higher: all messages  
  Note: the logging verbosity is a global property shared across multiple instances of the transport. If you create a new transport instance with a verbosity different than -1, the change will affect all the other instances as well. |
| security_verbosity                                      | Specifies the verbosity of security related log messages from the transport. These are usually messages generated by OpenSSL. Possible values:  
  - -1: silent  
  - 0: errors only  
  - 1: errors and warnings  
  - 2: local status  
  - 5 or higher: all messages  
  Default: If not set, the value is inherited from the verbosity property  
  Note: the logging verbosity is a global property shared across multiple instances of the transport. If you create a new transport instance with a security_verbosity different than -1, the change will affect all the other instances as well. |
| parent.parent.properties_bitmap                         | A bitmap that defines various properties of the transport to the Connext DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connext DDS tries to receive a message using the plugin. This is in support of a zero-copy interface. |

1 Assuming you used ‘dds.transport.WAN.wan1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ‘dds.transport.’
Table 42.1 Properties for NDDS_Transport_WAN_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.WAN.wan1.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.parent.gather_send_buffer_count_max</td>
<td>Specifies the maximum number of buffers that Connext DDS can pass to the send() function of the transport plugin. The transport plugin send() API supports a gather-send concept, where the send() call can take several discontiguous buffers, assemble and send them in a single message. This enables Connext DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer. However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connext DDS from trying to gather too many buffers into a send call for the transport plugin. Connext DDS requires all transport-plugin implementations to support a gather-send of at least a minimum number of buffers. This minimum number is defined as NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.</td>
</tr>
<tr>
<td>parent.parent.message_size_max</td>
<td>The maximum size of a message in bytes that can be sent or received by the transport plugin. Above this size, DDS-level fragmentation will occur. See 23.3 Large Data Fragmentation on page 970. This value must be set before the transport plugin is registered, so that Connext DDS can properly use the plugin. Default for Integrity platforms: 9216 Default for non-Integrity platforms: 65507</td>
</tr>
<tr>
<td>parent.parent.allow_interfaces</td>
<td>A list of strings, each identifying a range of interface addresses. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. If the list is non-empty, this &quot;white&quot; list is applied before the parent.parent.deny_interfaces on the next page list. It is up to the transport plugin to interpret the list of strings passed in. Usually this interpretation will be consistent with NDDS_Transport_String_To_Address_Fcn_cEA(). This property is not interpreted by the Connext DDS core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user. You must manage the memory of the list. The memory may be freed after the DomainParticipant is enabled.</td>
</tr>
</tbody>
</table>

1 Assuming you used ‘dds.transport.WAN.wan1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 42.1 Properties for NDDS_Transport_WAN_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.WAN.wan1.')</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.parent.deny_interfaces</td>
<td>A list of strings, each identifying a range of interface addresses. If the list is non-empty, deny the use of these interfaces. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. This &quot;black&quot; list is applied after the parent.parent.allow_interfaces on the previous page list and filters out the interfaces that should not be used. It is up to the transport plugin to interpret the list of strings passed in. Usually this interpretation will be consistent with NDDS_Transport_String_To_Address_Fcn_cEA(). This property is not interpreted by the Connext DDS core; it is provided merely as a convenient and standardized way to specify the interfaces for the benefit of the transport plugin developer and user. You must manage the memory of the list. The memory may be freed after the DomainParticipant is enabled.</td>
</tr>
<tr>
<td>parent.send_socket_buffer_size</td>
<td>Size in bytes of the send buffer of a socket used for sending. On most operating systems, setsockopt() will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to parent.parent.message_size_max on the previous page. The maximum value is operating system-dependent. If -1, setsockopt() (or equivalent) will not be called to size the send buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td>parent.recv_socket_buffer_size</td>
<td>Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to parent.parent.message_size_max on the previous page. The maximum value is operating system-dependent. If -1, setsockopt() (or equivalent) will not be called to size the receive buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td>parent.unicast_enabled</td>
<td>Allows the transport plugin to use unicast UDP for sending and receiving. By default, it will be turned on. Also by default, it will use all the allowed network interfaces that it finds up and running when the plugin is instantiated.</td>
</tr>
</tbody>
</table>
| parent.ignore_loopback_interface                      | Prevents the transport plugin from using the IP loopback interface. Three values are allowed:  
  - 0: Enable local traffic via this plugin. This plugin will only use and report the IP loopback interface only if there are no other network interfaces (NICs) up on the system.  
  - 1: Disable local traffic via this plugin. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plugin like Shared Memory instead of the IP loopback. |

1 Assuming you used ‘dds.transport.WAN.wan1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ‘dds.transport.’
### Table 42.1 Properties for NDDS_Transport_WAN_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
</table>
| parent.ignore_nonrunning_interfaces | Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed:  
  - 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.  
  - 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.  
  
By default this property is set to 1, so Connext DDS will ignore non-running interfaces. |
| DEPRECATED parent.no_zero_copy | Prevents the transport plugin from doing a zero copy. By default, this plugin will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application. The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the H/W, device driver, or the OS to allow the zero copy feature to work for your application, you may have no choice but to turn off zero copy use. By default this is set to 0, so Connext DDS will use the zero-copy API if offered by the OS. |
| parent.send_blocking | Controls the blocking behavior of send sockets. Two values are defined (use the number values only):  
  - 1 (NDDS_TRANSPORT_UDP_BLOCKING_ALWAYS): Sockets are blocking (default socket options for Operating System).  
  - 0 (NDDS_TRANSPORT_UDP_BLOCKING_NEVER): Sockets are modified to make them non-blocking. THIS IS NOT A SUPPORTED CONFIGURATION AND MAY CAUSE SIGNIFICANT PERFORMANCE PROBLEMS. |

---

1. Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
Table 42.1 Properties for NDDS_Transport_WAN_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.WAN.wan1.')</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.transport_priority_mask</td>
<td>Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low/high to define the mapping from DDS transport priority to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket. For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high. If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets.</td>
</tr>
<tr>
<td>parent.transport_priority_mapping_low</td>
<td>Sets the low and high values of the output range to IPv4 TOS. These values are used in conjunction with transport_priority_mask to define the mapping from DDS transport priority to the IPv4 TOS field. Defines the low and high values of the output range for scaling. Note that IPv4 TOS is generally an 8-bit value.</td>
</tr>
<tr>
<td>parent.transport_priority_mapping_high</td>
<td>Required if you want to use security.</td>
</tr>
<tr>
<td>enable_security</td>
<td>Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data. The minimum value for this property is parent.parent.message_size_max on page 1127. Default: 65507</td>
</tr>
<tr>
<td>recv_decode_buffer_size</td>
<td>Port offset to allow coexistence with non-secure UDP transport.</td>
</tr>
<tr>
<td>port_offset</td>
<td>DTLS handshake retransmission interval in milliseconds. Default: 1000</td>
</tr>
<tr>
<td>dtls_handshake_resend_interval</td>
<td>Liveliness interval (multiple of resend interval) The connection will be dropped if no message from the peer is received in this amount of time. This enables cleaning up state for peers that are no longer responding. A secure keep-alive message will be sent every half-interval if no other sends have occurred for a given DTLS connection during that time. Default: 60</td>
</tr>
<tr>
<td>dtls_connection_liveliness_interval</td>
<td>A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for SSL_load_verify_locations for more information. If you want to use security, tls.verify.ca_file above or tls.verify.ca_path on the next page must be specified; both may be specified.</td>
</tr>
<tr>
<td>tls.verify.ca_file</td>
<td></td>
</tr>
</tbody>
</table>

1 Assuming you used 'dds.transport.WAN.wan1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls.verify.ca_path</td>
<td>A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format, and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for SSL_CTX_bad_verify_locations for more information. The Certificate Authority subject name hash values must be available in the directories. You may generate them by running opensslrehash (available in OpenSSL 1.1.0 or above) or <em>c_rehash</em> in each directory. If you want to use security, <code>tls.verify.ca_file on the previous page</code> or <code>tls.verify.ca_path above</code> must be specified; both may be specified.</td>
</tr>
<tr>
<td>tls.verify.verify_depth</td>
<td>Maximum certificate chain length for verification.</td>
</tr>
<tr>
<td>tls.verify.verify_peer</td>
<td>If non-zero, use mutual authentication when performing TLS hand-shake (default). If zero, only the reader side will present a certificate, which will be verified by the writer side.</td>
</tr>
<tr>
<td>tls.verify.callback</td>
<td>This can be set to one of three values:</td>
</tr>
<tr>
<td></td>
<td>• &quot;default&quot; selects NDDS_Transport_TLS_default_verify_callback()</td>
</tr>
<tr>
<td></td>
<td>• &quot;verbose&quot; selects NDDS_Transport_TLS_verbose_verify_callback()</td>
</tr>
<tr>
<td></td>
<td>• &quot;none&quot; requests no callback be registered</td>
</tr>
<tr>
<td>tls.cipher.cipher_list</td>
<td>List of available DTLS ciphers when communicating with Connext DDS 6.0.0 or below. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string. Default: NULL.</td>
</tr>
<tr>
<td>tls.cipher.dh_param_files</td>
<td>List of available Diffie-Hellman (DH) key files. For example: &quot;foo.h:2048,bar.h:1024&quot; means: dh_param_files[0].file = foo.pem, dh_param_files[0].bits = 2048, dh_param_files[1].file = bar.pem, dh_param_files[1].bits = 1024</td>
</tr>
<tr>
<td>tls.cipher.engine_id</td>
<td>String ID of OpenSSL cipher engine to request.</td>
</tr>
<tr>
<td>tls.identity.certificate_chain_file</td>
<td>A string that specifies the name of a file containing an identifying certificate chain (in PEM format). An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If no private key is specified, this file will be used to load a non-RSA private key.</td>
</tr>
<tr>
<td>tls.identity.private_key_password</td>
<td>A string that specifies the password for private key.</td>
</tr>
<tr>
<td>tls.identity.private_key_file</td>
<td>A string that specifies that name of a file containing private key (in PEM format). If no private key is specified (all values are NULL), this value will default to the same file as the specified certificate chain file.</td>
</tr>
<tr>
<td>tls.identity.rsa_private_key_file</td>
<td>A string that specifies that name of a file containing an RSA private key (in PEM format).</td>
</tr>
</tbody>
</table>

1 Assuming you used `dds.transport.WAN.wan1` as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with `dds.transport.`
### Table 42.1 Properties for NDDS_Transport_WAN_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.WAN.wan1.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport_instance_id[0] to [NDDS_TRANS__TRANSPORT__TRANSPORT_INSTANCE_ID_LENGTH][0]</td>
<td>Required A set of comma-separated values to specify the elements of the array. This value must be unique for all transport instances communicating with the same WAN Rendezvous Server. If less than the full array is specified, it will be right-aligned. For example, the string &quot;01,02&quot; results in the array being set to: {0,0,0,0,0,0,0,0,0,0,1,2}</td>
</tr>
<tr>
<td>interface_address</td>
<td>Locator, as a string</td>
</tr>
<tr>
<td>server</td>
<td>Required Server locator, as a string.</td>
</tr>
<tr>
<td>server_port</td>
<td>Server port number.</td>
</tr>
<tr>
<td>stun_retransmission_interval</td>
<td>STUN request messages requiring a response are resent with this interval. The interval is doubled after each retransmission. Specified in msec.</td>
</tr>
<tr>
<td>stun_number_of_retransmissions</td>
<td>Maximum number of times STUN messages are resent unless a response is received.</td>
</tr>
<tr>
<td>stun_liveliness_period</td>
<td>Period at which messages are sent to peers to keep NAT holes open; and to the WAN server to refresh bound ports. Specified in msec.</td>
</tr>
<tr>
<td>parent.join_multicast_group_timeout</td>
<td>Windows only. On Windows, a network interface may be detected before it is allowed to join a multicast group address. This property adjusts how much time (in milliseconds) to wait for the ADD_MEMBERSHIP multicast operation to succeed before withdrawing. Default: 5000</td>
</tr>
</tbody>
</table>

---

1 Assuming you used ‘dds.transport.WAN.wan1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

2 NDDS_TRANS_\_TRANSPORT_\_TRANSPORT_INSTANCE_ID_LENGTH = 12
Table 42.1 Properties for NDDS_Transport_WAN_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.WAN.wan1.’)</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property_validation_action</td>
<td>By default, property names given in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the transport:</td>
</tr>
<tr>
<td></td>
<td>• VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail.</td>
</tr>
<tr>
<td></td>
<td>• VALIDATION_ACTION_SKIP: skip validation.</td>
</tr>
<tr>
<td></td>
<td>• VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail.</td>
</tr>
<tr>
<td></td>
<td>If this property is not set, the property validation behavior will be the same as that of the DomainParticipant, which by default is VALIDATION_ACTION_EXCEPTION. See 7.5.19.1 Property Validation on page 441 for more information.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>thread_name_prefix</th>
<th>You can set this field with your own value, to help you identify the transport thread in a way that's meaningful to you. Do not exceed 8 characters.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>If you do not set this field, Connext DDS creates the following prefix:</td>
</tr>
<tr>
<td></td>
<td>‘Y’ + ‘Tr’ + participant identifier + ‘0’</td>
</tr>
<tr>
<td></td>
<td>Where ‘Y’ indicates this is a thread from RTI, ‘Tr’ indicates the thread is related to a transport, and participant identifier contains 5 characters as follows:</td>
</tr>
<tr>
<td></td>
<td>• If participant_name is set: The participant identifier will be the first 3 characters and the last 2 characters of the participant_name.</td>
</tr>
<tr>
<td></td>
<td>• If participant_name is not set, then the identifier is computed as domain_id (3 characters) followed by participant_id (2 characters).</td>
</tr>
<tr>
<td></td>
<td>• If participant_name is not set and the participant_id is set to -1 (default value), then the participant identifier is computed as the last 5 digits of the rtps_instance_id in the participant GUID.</td>
</tr>
</tbody>
</table>

See 21.9 Identifying Threads Used by Connext DDS on page 916.

42.4 Secure Transport Properties

Table 42.2 Properties for NDDS_Transport_DTLS_Property_t lists the properties that you can set for the Secure Transport.

---

1 Assuming you used ‘dds.transport.WAN.wan1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 42.2 Properties for NDDS_Transport_DTLS_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.DTLS.dtls1')&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.transport.load_plugins (note: this does not take a prefix)</td>
<td><strong>Required</strong>&lt;br&gt;Comma-separated strings indicating the prefix names of all plugins that will be loaded by Connext DDS. You will use this string as the prefix to the property names. For example: “dds.transport.DTLS.dtls1” (This assumes you used used ‘dds.transport.DTLS.dtls1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins.) This prefix must begin with ‘dds.transport.’ Note: you can load up to 8 plugins.</td>
</tr>
<tr>
<td>library</td>
<td><strong>Only required if linking dynamically</strong>&lt;br&gt;Must set to “libnddstransporttls.so” (for Linux systems) or “nddstransporttls.dll” (for Windows systems). This library and the dependent OpenSSL libraries must be in your library search path (pointed to by the environment variable LD_LIBRARY_PATH on Linux systems, Path on Windows systems, DYLD_LIBRARY_PATH on macOS systems).</td>
</tr>
<tr>
<td>create_function</td>
<td><strong>Only required if linking dynamically</strong>&lt;br&gt;Must be “NDDS_Transport_DTLS_create”</td>
</tr>
<tr>
<td>create_function_ptr</td>
<td><strong>Only required if linking statically</strong>&lt;br&gt;Defines the function pointer to the DTLS Transport Plugin creation function. Used for loading the DTLS Transport plugin statically. Must be set to the NDDS_Transport_DTLS_create function pointer.</td>
</tr>
<tr>
<td>aliases</td>
<td>Used to register the transport plugin returned by NDDS_Transport_DTLS_create() (as specified by &lt;DTLS_prefix&gt;.create_function) to the DomainParticipant. Aliases should be specified as comma-separated strings, with each comma delimiting an alias. If it is not specified, the prefix—without the leading “dds.transport”—is used as the default alias for the plugin. For example, if the &lt;TRANSPORT_PREFIX&gt; is “dds.transport.mytransport”, the default alias for the plugin is “mytransport”.</td>
</tr>
<tr>
<td>network_address</td>
<td>The network address at which to register this transport plugin. The least significant transport_in.property.address_bit_count will be truncated. The remaining bits are the network address of the transport plugin. This value overwrites the value returned by the output parameter in NDDS_Transport_create_plugin function as specified in &quot;&lt;DTLS_prefix&gt;.create_function&quot;.</td>
</tr>
</tbody>
</table>

---

<sup>a</sup> Assuming you used ‘dds.transport.DTLS.dtls1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ‘dds.transport.’
<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.DTLS.dtls1')</th>
<th>Property Value Description</th>
</tr>
</thead>
</table>
| verbosity                                              | Specifies the verbosity of log messages from the transport. Possible values:  
  - 1: silent  
  - 0: errors only  
  - 1: errors and warnings  
  - 2: local status  
  - 5 or higher: all messages  
Note: the logging verbosity is a global property shared across multiple instances of the transport. If you create a new transport instance with a verbosity different than -1, the change will affect all the other instances as well. |
| security_verbosity                                     | Specifies the verbosity of security related log messages from the transport. These are usually messages generated by OpenSSL. Possible values:  
  - 1: silent  
  - 0: errors only  
  - 1: errors and warnings  
  - 2: local status  
  - 5 or higher: all messages  
Default: If not set the value is inherited from the property verbosity. Note: the logging verbosity is a global property shared across multiple instances of the transport. If you create a new transport instance with a security_verbosity different than -1, the change will affect all the other instances as well. |
| parent.properties_bitmap                               | A bitmap that defines various properties of the transport to the Connext DDS core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when Connext DDS tries to receive a message using the plugin. This is in support of a zero-copy interface. |
| parent.gather_send_buffer_count_max                    | Specifies the maximum number of buffers that Connext DDS can pass to the transport plugin’s send() function. |
| parent.message_size_max                                | The maximum size of a message in bytes that can be sent or received by the transport plugin. Above this size, DDS-level fragmentation will occur. See 23.3 Large Data Fragmentation on page 970. Default and maximum value: 16384. If you attempt to exceed 16384, then the transport will log a warning message and automatically adjust message_size_max to be 16384. |

---

a Assuming you used ‘dds.transport.DTLS.dtls1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
42.4 Secure Transport Properties

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>parent.allow_interfaces</td>
<td>A list of strings, each identifying a range of interface addresses. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. If the list is non-empty, this “white” list is applied before the parent.deny_interfaces list. You must manage the memory of the list. The memory may be freed after the DomainParticipant is enabled.</td>
</tr>
<tr>
<td>parent.deny_interfaces</td>
<td>A list of strings, each identifying a range of interface addresses. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. This “black” list is applied after the parent.allow_interfaces list and filters out the interfaces that should not be used. You must manage the memory of the list. The memory may be freed after the DomainParticipant is enabled.</td>
</tr>
<tr>
<td>disable_interface_tracking</td>
<td>Disables detection of network interface changes. By default, network interfaces changes are propagated in the form of locators to other applications. This is done to support IP mobility scenarios. For example, you could start a application with Wi-Fi and move to a wired connection. In order to continue communicating with other applications this interface change must be propagated. In 5.0 and earlier versions of the product, IP mobility scenarios were not supported. Applications using 5.2 will report errors if they detect locator changes in a DataWriter or DataReader. You can disable the notification and propagation of interface changes by setting this property to 1. This way, an interface change in a newer application will not trigger errors in an application running 5.2 GA or earlier. Of course, this will prevent the new application from being able to detect network interface changes.</td>
</tr>
<tr>
<td>interface_poll_period</td>
<td>Specifies the period in milliseconds to query for changes in the state of all the interfaces. When possible, the detection of an IP address changes is done asynchronously using the APIs offered by the underlying OS. If there is no mechanism to do that, the detection will use a polling strategy where the polling period can be configured by setting this property.</td>
</tr>
<tr>
<td>send_socket_buffer_size</td>
<td>Size in bytes of the send buffer of a socket used for sending. On most operating systems, setsockopt() will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to parent.message_size_max. The maximum value is operating system-dependent. If -1, setsockopt() (or equivalent) will not be called to size the send buffer of the socket. The transport will use the OS default. Default: 131072.</td>
</tr>
<tr>
<td>recv_socket_buffer_size</td>
<td>Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to parent.message_size_max. The maximum value is operating system-dependent. If -1, setsockopt() (or equivalent) will not be called to size the receive buffer of the socket. The transport will use the OS default. Default: 131072.</td>
</tr>
</tbody>
</table>

---

*Assuming you used ‘dds.transport.DTLS.dtls1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.*'
42.4 Secure Transport Properties

Table 42.2 Properties for NDDS_Transport_DTLS_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.DTLS.dtls1')</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ignore_loopback_interface</td>
<td>Prevents the Transport Plugin from using the IP loopback interface.</td>
</tr>
<tr>
<td>ignore_nonrunning_interfaces</td>
<td>Prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that &quot;all resources are allocated&quot;, and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed: 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.</td>
</tr>
<tr>
<td>transport_priority_mask</td>
<td>Mask for use of transport priority field.</td>
</tr>
<tr>
<td>transport_priority_mapping_low</td>
<td>Low and high values of output range to IPv4 TOS.</td>
</tr>
<tr>
<td>transport_priority_mapping_high</td>
<td></td>
</tr>
<tr>
<td>recv_decode_buffer_size</td>
<td>Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data. The minimum value for this property is parent.message_size_max on page 1135. Default: 65507</td>
</tr>
<tr>
<td>port_offset</td>
<td>Port offset to allow coexistence with non-secure UDP transport.</td>
</tr>
<tr>
<td>dtls_handshake_resend_interval</td>
<td>DTLS handshake retransmission interval in milliseconds Default: 1000</td>
</tr>
<tr>
<td>dtls_connection_liveliness_interval</td>
<td>Liveliness interval (multiple of resend interval) The connection will be dropped if no message from the peer is received in this amount of time. This enables cleaning up state for peers that are no longer responding. A secure keep-alive message will be sent every half-interval if no other sends have occurred for a given DTLS connection during that time. Default: 60</td>
</tr>
<tr>
<td>tls.verify.ca_file</td>
<td>A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for SSL_load_verify_locations for more information. tls_verify.ca_file above or tls_verify.ca_path on the next page must be specified; both may be specified.</td>
</tr>
</tbody>
</table>

\[\text{a} \quad \text{Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'}\]
### Table 42.2 Properties for NDDS_Transport_DTLS_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.DTLS.dtls1')&lt;sup&gt;a&lt;/sup&gt;</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls.verify.ca_path</td>
<td>A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format, and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for SSL_CTX_load_verify_locations for more information. The Certificate Authority subject name hash values must be available in the directories. You may generate them by running openssl_rehash (available in OpenSSL 1.1.0 or above) or <em>c_rehash</em> in each directory. tls.verify.ca_file on the previous page or tls.verify.ca_path above must be specified; both may be specified.</td>
</tr>
<tr>
<td>tls.verify.verify_depth</td>
<td>Maximum certificate chain length for verification.</td>
</tr>
<tr>
<td>tls.verify.verify_peer</td>
<td>If non-zero, use mutual authentication when performing TLS hand-shake (default). If zero, only the reader side will present a certificate, which will be verified by the writer side.</td>
</tr>
</tbody>
</table>
| tls.verify.callback                                           | This can be set to one of three values:

- "default" selects NDDS_Transport_TLS_default_verify_callback()
- "verbose" selects NDDS_Transport_TLS_verbose_verify_callback()
- "none" requests no callback be registered |
| tls.cipher.cipher_list                                        | List of available DTLS ciphers when communicating with Connext DDS 6.0.0 or below. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string. Default: NULL |
| tls.cipher.dh_param_files                                     | List of available Diffie-Hellman (DH) key files. For example: "foo.h:2048,bar.h:1024" means:

- `dh_param_files[0].file = foo.pem`,
- `dh_param_files[0].bits = 2048`,
- `dh_param_files[1].file = bar.pem`,
- `dh_param_files[1].bits = 1024` |
| tls.cipher.engine_id                                          | String ID of OpenSSL cipher engine to request. |
| tls.identity.certificate_chain_file                           | Required A string that specifies the name of a file containing an identifying certificate chain (in PEM format). An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If no private key is specified, this file will be used to load a non-RSA private key. |
| tls.identity.private_key_password                             | A string that specifies the password for private key. |
| tls.identity.private_key_file                                 | A string that specifies that name of a file containing private key (in PEM format). If no private key is specified (all values are NULL), this value will default to the same file as the specified certificate chain file. |
| tls.identity.rsa_private_key_file                             | A string that specifies that name of a file containing an RSA private key (in PEM format). |

<sup>a</sup> Assuming you used ‘dds.transport.DTLS.dtls1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
Table 42.2 Properties for NDDS_Transport_DTLS_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.DTLS.dtls1')a</th>
<th>Property Value Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>join_multicast_group_timeout</td>
<td>Windows only. On Windows, a network interface may be detected before it is allowed to join a multicast group address. This property adjusts how much time (in milliseconds) to wait for the ADD_MEMBERSHIP multicast operation to succeed before withdrawing. Default: 5000</td>
</tr>
</tbody>
</table>
| property_validation_action                               | By default, property names given in the [7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439](#) are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the transport:  
  - VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail.  
  - VALIDATION_ACTION_SKIP: skip validation.  
  - VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail.  
  If this property is not set, the property validation behavior will be the same as that of the DomainParticipant, which by default is VALIDATION_ACTION_EXCEPTION. See [7.5.19.1 Property Validation on page 441](#) for more information. |
| thread_name_prefix                                       | You can set this field with your own value, to help you identify the transport thread in a way that's meaningful to you. Do not exceed 8 characters. If you do not set this field, Connext DDS creates the following prefix:  
  `Y + 'Tr' + participant identifier + '0'`  
  Where 'Y' indicates this is a thread from RTI, 'Tr' indicates the thread is related to a transport, and participant identifier contains 5 characters as follows:  
  - If `participant_name` is set: The participant identifier will be the first 3 characters and the last 2 characters of the `participant_name`.  
  - If `participant_name` is not set, then the identifier is computed as `domain_id` (3 characters) followed by `participant_id` (2 characters).  
  - If `participant_name` is not set and the `participant_id` is set to -1 (default value), then the participant identifier is computed as the last 5 digits of the `rtps_instance_id` in the participant GUID.  
  See [21.9 Identifying Threads Used by Connext DDS on page 916](#). |

42.5 Explicitly Instantiating a WAN or Secure Transport Plugin

As described on [Page 1122](#), there are two ways to instantiate a transport plugin. This section describes the mechanism that includes calling `NDDSTransportSupport::register_transport()`. (The other way is to use the Property QoS mechanism, described in [42.2 Setting Up a Transport with the Property QoS on page 1123](#)).

---

*a Assuming you used ‘dds.transport.DTLS.dtls1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
Notes:

- This way of instantiating a transport is not supported in the Java API. If you are using Java, use the Property QoS mechanism, described in 42.2 Setting Up a Transport with the Property QoS on page 1123.

- To use this mechanism, there are **extra libraries that you must link into your program and an additional header file** that you must include. Please see the 42.5.1 Additional Header Files and Include Directories below and 42.5.2 Additional Libraries below for details.

To instantiate a WAN or Secure Transport prior to explicitly registering it with `NDDS_TransportSupport::register_transport()`, use one of the following functions:

```c
NDDS_Transport_Plugin* NDDS_Transport_WAN_new (const struct NDDS_Transport_WAN_Property_t * property_in)
NDDS_Transport_Plugin* NDDS_Transport_DTLS_new (const struct NDDS_Transport_DTLS_Property_t * property_in)
```

See the API Reference HTML documentation for details on these functions.

### 42.5.1 Additional Header Files and Include Directories

- To use the **Secure WAN Transport** API, you must include an extra header file (in addition to those in **Table 10.1 Header Files to Include for Connext DDS (All Architectures)**).

```c
#include "ndds/ndds_transport_secure_wan.h"
```

Assuming that **Secure WAN Transport** is installed in the same directory as **Connext DDS** (see **Table 10.2 Include Paths for Compilation (All Architectures)**), no additional include paths need to be added for the **Secure WAN Transport** API. If this is not the case, you will need to specify the appropriate include path.

- If you want to access OpenSSL data structures, add the OpenSSL include directory, `<openssl install dir>/arch/include`, and include the OpenSSL headers **before** `ndds_transport_secure_wan.h`:

```c
#include <openssl/ssl.h>
#include <openssl/x509.h> (if accessing certificate functions)
```

**On Windows systems, if you are loading statically**: you should also include the OpenSSL file, `applink.c`, in your application. It can be found in the OpenSSL include directory, or included as `<openssl/applink.c>`.

### 42.5.2 Additional Libraries

To use the **Secure WAN Transport** API, you must link in additional libraries, which are listed in the RTI Connext DDS Core Libraries Platform Notes (in the appropriate section for your architecture). Refer to
10.3.1 Required Libraries on page 682 for the differences between shared and static libraries.

42.5.3 Compiler Flags

No additional compiler flags are required.
Part 7: RTI TCP Transport

RTI TCP Transport is only available on specific architectures. See the RTI Connext DDS Core Libraries Platform Notes for details.

Out of the box, Connext DDS uses the UDPv4 and Shared Memory transport to communicate with other DDS applications. This configuration is appropriate for systems running within a single LAN. However, using UDPv4 introduces some problems when Connext DDS applications in different LANs need to communicate:

- UDPv4 traffic is usually filtered out by the LAN firewalls for security reasons.
- Forwarded ports are usually TCP ports.
- Each LAN may run in its own private IP address space and use NAT (Network Address Translation) to communicate with other networks.

TCP Transport enables participant discovery and data exchange using the TCP protocol (either on a local LAN, or over the public WAN). TCP Transport allows Connext DDS to address the challenges of using TCP as a low-level communication mechanism between peers and limits the number of ports exposed to one. (When using the default UDP transport, a Connext DDS application uses multiple UDP ports for communication, which may make it unsuitable for deployment across firewalled networks).
Chapter 43 TCP Communication Scenarios

TCP Transport can be used to address multiple communication scenarios—from simple communication within a single LAN, to complex communication scenarios across LANs where NATs and firewalls may be involved. This section describes these scenarios:

- 43.1 Communication Within a Single LAN below
- 43.2 Symmetric Communication Across NATs on the next page
- 43.3 Asymmetric Communication Across NATs on page 1145

43.1 Communication Within a Single LAN

TCP Transport can be used as an alternative to UDPv4 to communicate with Connext DDS applications running inside the same LAN. Figure 43.1: Communication within a Single LAN on the next page shows how to configure the TCP transport in this scenario.
43.2 Symmetric Communication Across NATs

In NAT communication scenarios, each one of the LANs has a private IP address space. The communication with other LANs is done through NAT routers that translate private IP addresses and ports into public IP addresses and ports.

In symmetric communication scenarios, any Connext DDS application can initiate TCP connections with other applications. Figure 43.2: Symmetric Communication Across NATs on the next page shows how to configure the TCP transport in this scenario.

- **parent.classid** on page 1160 and **server_bind_port** on page 1165 are transport properties configured using the PropertyQosPolicy of the participant. (Note: When the TCP transport is instantiated, by default it is configured to work in a LAN environment using symmetric communication and binding to port 7400 for incoming connections.) For additional information about these properties, see Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t.

- Initial Peers represents the peers to which the participant will be announced to. Usually, these peers are configured using the DiscoveryQosPolicy of the participant or the environment variable NDDS_DISCOVERY_PEERS. For information on the format of initial peers, see 44.1 Choosing a Transport Mode on page 1148.

Unlike the UDPv4 transport, **you must specify the initial peers**, because multicast cannot be used with TCP.
Notice that initial peers refer to the public address of the remote LAN where the Connext DDS application is deployed and not the private address of the node where the application is running. In addition, the transport associated with a Connext DDS instance will have to be configured with its public address (public_address on page 1164) so that this information can be propagated as part of the discovery process.

Because the public address and port of the Connext DDS instances must be known before the communication is established, the NAT Routers will have to be configured statically to translate (forward) the private server_bind_port on page 1165 into a public port. This process is known as static NAT or port forwarding; it allows traffic originating in outer networks to reach designated peers in the LAN behind the NAT router. You will need to refer to your router’s configuration manual to understand how to correctly set up port forwarding.

### 43.3 Asymmetric Communication Across NATs

This scenario is similar to the previous one, except in this case the TCP connections can be initiated only by the Connext DDS instance in LAN1. For security reasons, incoming connections to LAN1 are not allowed. In this case, the peer in LAN1 is considered ‘unreachable.’ Unreachable peers can publish and subscribe just like any other peer, but communication can occur only to a ‘reachable’ peer.
Figure 43.3: Asymmetric Communication Across NATs below shows how to configure the TCP transport in this scenario. Notice that the transport property `server_bind_port` is set to 0 to configure the node as unreachable.

**Figure 43.3: Asymmetric Communication Across NATs**

In an asymmetric configuration, an unreachable peer (that is behind a firewall or NAT without port forwarding) can still publish and subscribe like a reachable peer, but with some important limitations:

- An unreachable peer can only communicate with reachable peers: two unreachable peers cannot establish a direct communication since they are both behind a firewall and/or NAT.

Note that since Connext DDS always relies on a direct connection between peers (even if there is a third node that can be reachable by both unreachable peers), **communication can never occur between unreachable peers**. For example, suppose Peers A and B are unreachable and Peer C is reachable. Communication can take place between A and C, and between B and C, but not between A and B.

- It can take longer to discover unreachable peers than reachable ones. This is because a reachable peer has to wait for the unreachable peer to establish the communication first.

For example, suppose Peer A (unreachable) starts before Peer B (reachable). The discovery mechanism of A attempts to connect to the (not-yet existing) Peer B. Since it fails, it will retry after \( n \) seconds. Right after that, B starts. If A would be reachable (and in B’s peer list), the discovery
mechanism will immediately contact A. In this case, since A cannot be reached, B needs to wait until the discovery process of A decides to retry.

This effect can be minimized by modifying the QoS that controls the discovery mechanism used by A. In particular, you should set the DomainParticipant’s DiscoveryConfig QoS policy’s min_initial_participant_announcement_period to a small value.

Note that the concept of symmetric/asymmetric configuration is a local concept that only describes the communication mechanism between two peers. A reachable peer can be involved in symmetric communication with another reachable peer, and at the same time have asymmetric communication with a unreachable peer. When a peer attempts to communicate with a remote peer, it knows if the remote peer is reachable or not by looking at the transport address provided.
Chapter 44 Configuring the TCP Transport

TCP Transport is distributed as a both shared and static library in `<NDDSHOME>/lib/<architecture>`. The library is called `nddstransporttcp`.

Mechanisms for Configuring the Transport:

- **By explicitly instantiating a new transport** (see 44.2 Explicitly Instantiating the TCP Transport Plugin on the next page) and then registering it with the DomainParticipant (see 16.7 Installing Additional Builtin Transport Plugins with register_transport() on page 821). (Not available in the Java and .NET APIs.)

- **Through the Property QoS policy** of the DomainParticipant (on Linux and Windows systems only). This process is described in 44.3 Configuring the TCP Transport with the Property QosPolicy on page 1151.

This section describes:

- 44.1 Choosing a Transport Mode below
- 44.2 Explicitly Instantiating the TCP Transport Plugin on the next page
- 44.3 Configuring the TCP Transport with the Property QosPolicy on page 1151
- 44.4 Setting the Initial Peers on page 1154
- 44.6 Support for External Hardware Load Balancers in TCP Transport Plugin on page 1156
- 44.7 TCP/TLS Transport Properties on page 1158

44.1 Choosing a Transport Mode

When you configure the TCP transport, you must choose one of the following types of communication:
44.2 Explicitly Instantiating the TCP Transport Plugin

- **TCP over LAN** — Communication between the two peers is not encrypted (data is written directly to a TCP socket). Each node can use all the possible interfaces available on that machine to receive connections. The node can only receive connections from machines that are on a local LAN.

- **TCP over WAN** — Communication is not encrypted (data is written directly to a TCP socket). The node can only receive connections from a specific port, which must be configured in the public router of the local network (WAN mode).

- **TLS over LAN** — This is similar to the TCP over LAN, where the node can use all the available network interfaces to TX/RX data (LAN nodes only), but in this mode, the data being written on the physical socket is encrypted first (through the openssl library). Performance (throughput and latency) may be less than TCP over LAN since the data needs to be encrypted before going on the wire. Discovery time may be longer with this mode because when the first connection is established, the two peers exchange handshake information to ensure line protection. For more general information on TLS, see 41.3 Datagram Transport-Layer Security (DTLS) on page 1116.

- **TLS over WAN** — The data is encrypted just like TLS over LAN, but it can be sent and received only from a specific port of the router.

**Note:** To use either TLS mode, you also need RTI TLS Support, which is available for purchase as a separate package. TLS Support uses TLS 1.3.

An instance of the transport can only communicate with other nodes that use the same transport mode.

You can specify the transport mode in either the NDDS_Transport_TCPv4_Property_t structure (see 44.7 TCP/TLS Transport Properties on page 1158) or in the parent.classid on page 1160 field of the Properties QoS (see 44.3 Configuring the TCP Transport with the Property QosPolicy on page 1151). Your choice of transport mode will also be reflected in the prefix you use for setting the initial peers (see 44.4 Setting the Initial Peers on page 1154).

### 44.2 Explicitly Instantiating the TCP Transport Plugin

As described on Page 1148, there are two ways to configure a transport plugin. This section describes the way that includes explicitly instantiating and registering a new transport. (The other way is to use the Property QoS mechanism, described in 44.3 Configuring the TCP Transport with the Property QosPolicy on page 1151).

**Notes:**

This way of instantiating a transport is not supported in the Java and .NET APIs. If you are using Java or .NET, use the Property QoS mechanism described in 44.3 Configuring the TCP Transport with the Property QosPolicy on page 1151.

To use this mechanism, there are extra libraries that you must link into your program and an additional header file that you must include. Please see 44.2.1 Additional Header Files and Include Directories on the next page and 44.2.2 Additional Libraries and Compiler Flags on the next page for details.
To instantiate a TCP transport:

Include the extra header file described in 44.2.1 Additional Header Files and Include Directories below.

Instantiate a new transport by calling `NDDS_Transport_TCPv4_new()`:

```c
NDDS_Transport_Plugin* NDDS_Transport_TCPv4_new
(const struct NDDS_Transport_TCPv4_Property_t *property_in)
```

Register the transport by calling `NDDSTransportSupport::register_transport()`.

See the API Reference HTML documentation for details on these functions and the contents of the NDDS_Transport_TCPv4_Property_t structure.

### 44.2.1 Additional Header Files and Include Directories

To use the TCP Transport API, you must include an extra header file (in addition to those in Table 10.1 Header Files to Include for Connext DDS (All Architectures)):

```c
#include "ndds/transport_tcp/transport_tcp_TCPv4.h"
```

Since TCP Transport is in the same directory as Connext DDS (see Table 10.2 Include Paths for Compilation (All Architectures)), no additional include paths need to be added for the TCP Transport API. If this is not the case, you will need to specify the appropriate include path.

### 44.2.2 Additional Libraries and Compiler Flags

To use the TCP Transport, you must add the `nddstransporttcp` library to the link phase of your application. There are four different kinds of libraries, depending on if you want a debug or release version, and static or dynamic linking with Connext DDS.

**Note:** Make sure your chosen kinds of libraries (static, dynamic, release, or debug) are consistent with the other Connext DDS libraries that your application links with. For example, if you are using RTI static core libraries, also use the static TCP Transport libraries. See 10.3.2 Mixing Static and Dynamic Libraries not Supported on page 682.

For Linux systems, the libraries are:

- `libnddstransporttcp.a` — Release version, dynamic libraries
- `libnddstransporttcpd.a` — Debug version, dynamic libraries
- `libnddstransporttcpz.a` — Release version, static libraries
- `libnddstransporttcpzd.a` — Debug version, static libraries

For Windows systems, the libraries are:
44.3 Configuring the TCP Transport with the Property QosPolicy

- NDDSTRA NPORTTCP.LIB — Release version, dynamic libraries
- NDDSTRA NPORTTCPD.LIB — Debug version, dynamic libraries
- NDDSTRA NPORTTCPZ.LIB — Release version, static libraries
- NDDSTRA NPORTTCPZD.LIB — Debug version, static libraries

Notes for using TLS:

To use either TLS mode (see 44.1 Choosing a Transport Mode on page 1148), you also need RTI TLS Support, which is available for purchase as a separate package. The TLS library (libnddstls.so or NDDSTLS.LIB, depending on your platform) must be in your library search path (pointed to by the environment variable LD_LIBRARY_PATH on Linux systems, Path on Windows systems, DYLD_LIBRARY_PATH on macOS systems).

If you already have SNDDSHOME/lib/<architecture> in your library search path, no extra steps are needed to use TLS once TLS Support is installed.

Even if you link everything statically, you must make sure that the location for SNDDSHOME/lib/<architecture> (or wherever the TLS library is located) is in your search path. When the TCP Transport Plugin is explicitly instantiated, the TLS library is loaded dynamically, even if you use static linking for everything else. To load TLS libraries statically, please see 44.3 Configuring the TCP Transport with the Property QosPolicy below.

Your search path must also include the location for the OpenSSL library, which is used by the TLS library.

44.3 Configuring the TCP Transport with the Property QosPolicy

The 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 allows you to set up name/value pairs of data and attach them to an entity, such as a DomainParticipant.

Like all QoS policies, there are two ways to specify the Property QoS policy:

Programmatically, as described in this section and 4.1.7 Getting, Setting, and Comparing QosPolicies on page 172. This includes using the add_property() operation to attach name/value pairs to the Property QosPolicy and then configuring the DomainParticipant to use that QosPolicy (by calling set_qos() or specifying QoS values when the DomainParticipant is created).

With an XML QoS Profile, as described in Configuring QoS with XML (Chapter 19 on page 848). This causes Connext DDS to dynamically load the TCP Transport library at runtime and then implicitly create and register the transport plugin.

Note: Dynamically load the TCP Transport library only if your application also links dynamically with the Connext DDS core libraries. See 10.3.2 Mixing Static and Dynamic Libraries not Supported on page 682.
To add name/value pairs to the Property QoS policy, use the add_property() operation:

```cpp
DDS_ReturnCode_t DDSPropertyQosPolicyHelper::add_property
    (DDS_PropertyQosPolicy policy, const char * name,
    const char * value, DDS_Boolean propagate)
```

For more information on add_property() and the other operations in the DDSPropertyQosPolicyHelper class, see Table 7.65 PropertyQoSPolicyHelper Operations, as well as the API Reference HTML documentation.

The ‘name’ part of the name/value pairs is a predefined string. The property names for the TCP Transport are described in Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t.

Here are the basic steps, taken from the example Hello World application (for details, please see the example application.)

Get the default DomainParticipant QoS from the DomainParticipantFactory.

```cpp
DDSDomainParticipantFactory::get_instance()->
    get_default_participant_qos(participant_qos);
```

Disable the builtin transports.

```cpp
participant_qos.transport_builtin.mask =
    DDS_TRANSPORTBUILTIN_MASK_NONE;
```

Set up the DomainParticipant’s Property QoS.

Load the plugin.

```cpp
DDSPropertyQosPolicyHelper::add_property (participant_qos.property,
    "dds.transport.load_plugins",
    "dds.transport.TCPv4.tcp1",
    DDS_BOOLEAN_FALSE);
```

Specify the transport plugin library.

```cpp
DDSPropertyQosPolicyHelper::add_property (participant_qos.property,
    "dds.transport.TCPv4.tcp1.library",
    "nddstransporttcp",
    DDS_BOOLEAN_FALSE);
```

Specify the transport’s ‘create’ function.

```cpp
DDSPropertyQosPolicyHelper::add_property (participant_qos.property,
    "dds.transport.TCPv4.tcp1.create_function",
    "NDDS_Transport_TCPv4_create",
    DDS_BOOLEAN_FALSE);
```

Set the transport to work in a WAN configuration with a public address:

```cpp
DDSPropertyQosPolicyHelper::add_property (participant_qos.property,
    "NDDS_Transport_TCPv4_public_address",
    "1152",
    DDS_BOOLEAN_FALSE);
```
44.3.1 Configuring the TCP Transport to be Loaded Statically

Similar to the previous example, here are the basic steps to load the TCP Transport plugin statically.

Note: Statically load the TCP Transport library only if your application also links statically with the Connext DDS core libraries. See 10.3.2 Mixing Static and Dynamic Libraries not Supported on page 682.

1. Get the default DomainParticipant QoS from the DomainParticipantFactory.

2. Disable the builtin transports.

3. Set up the DomainParticipant's Property QoS.
   a. Load the plugin.
   b. Specify the transport’s ‘create’ function pointer.

Property changes should be made before the transport is loaded—either before the DomainParticipant is enabled, before the first DataWriter/DataReader is created, or before the builtin topic reader is looked up, whichever one happens first.

Specify any other properties, as needed.

Create the DomainParticipant using the modified QoS.
c. Set the transport to work in a WAN configuration with a public address:

```c
DDSPropertyQosPolicyHelper::add_property
(participant_qos.property,
 "dds.transport.TCPv4.tcp1.parent.classid",
 "NDDS_TRANSPORT_CLASSID_TCPV4_WAN",
 DDS_BOOLEAN_FALSE);
DDSPropertyQosPolicyHelper::add_property
(participant_qos.property,
 "dds.transport.TCPv4.tcp1.public_address",
 "182.181.2.31",
 DDS_BOOLEAN_FALSE);
```

d. Specify any other properties, as needed.

4. Create the DomainParticipant using the modified QoS.

```c
participant = DDSTheParticipantFactory->create_participant
  (domainId, participant_qos,
   NULL /* listener */, DDS_STATUS_MASK_NONE);
```

### 44.3.2 Loading TLS Support Libraries Statically

The process to load TLS Support library statically is similar, but in this case both the `tls_create_function_ptr` and `tls_delete_function_ptr` properties need to be set.

```c
DDSPropertyQosPolicyHelper::add_pointer_property
  (participant_qos.property,
   "dds.transport.TCPv4.tcp1.tls_create_function_ptr",
   (void*)RTITLS_ConnectionEndpointFactoryTLSv4_create);
DDSPropertyQosPolicyHelper::add_pointer_property
  (participant_qos.property,
   "dds.transport.TCPv4.tcp1.tls_delete_function_ptr",
   (void*)RTITLS_ConnectionEndpointFactoryTLSv4_delete);
```

### 44.4 Setting the Initial Peers

**Note:** You must specify the initial peers (you cannot use the defaults because multicast cannot be used with TCP).

For TCP Transport, the addresses of the initial peers (NDDS_DISCOVERY_PEERS) that will be contacted during the discovery process have the following format:

- For WAN communication using TCP: `tcpv4_wan://<IP address or hostname>::<port>`
- For WAN communication using TLS: `tlsv4_wan://<IP address or hostname>::<port>`
44.5 RTPS Locator Format

- For LAN communication using TCP: `tcpv4_lan://<IP address or hostname>:<port>`
- For LAN communication using TLS: `tlsv4_lan://<IP address or hostname>:<port>`

For example (enter this on one line):

```bash
export NDDS_DISCOVERY_PEERS=
```

When the TCP transport is configured for LAN communication (with the `parent.classid` on page 1160 property), the IP address is the LAN address of the peer and the port is the server port used by the transport (the `server_bind_port` on page 1165 property).

When the TCP transport is configured for WAN communication (with the `parent.classid` on page 1160 property), the IP address is the WAN or public address of the peer and the port is the public port that is used to forward traffic to the server port in the TCP transport.

44.5 RTPS Locator Format

As described in Chapter 17 RTPS Locators and IP Mobility on page 827, an RTPS locator is an n-tuple (transport, address, port) that describes how to reach a remote endpoint.

The RTI TCP Transport locator has the following format:

```
<table>
<thead>
<tr>
<th>0</th>
<th>8</th>
<th>16</th>
<th>24</th>
<th>31</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

where kind can be one of the following values:

```c
#define NDDS_TRANSPORT_CLASSID_TCPV4_LAN (8)
#define NDDS_TRANSPORT_CLASSID_TCPV4_WAN (9)
#define NDDS_TRANSPORT_CLASSID_TLSV4_LAN (10)
#define NDDS_TRANSPORT_CLASSID_TLSV4_WAN (11)
```

There are two subkinds of RTI TCP locator, which differ in the way the address field is mapped. You can distinguish the two subkinds from each other by comparing bytes `address[8]` and `address[9]`:

- If `address[8]==0xFF` and `address[9]==0xFF`, the RTI TCP locator is an RTI TCP server locator, and the format of the address[16] is as follows:
Support for External Hardware Load Balancers in TCP Transport Plugin

For two Connext DDS applications to communicate, the TCP Transport Plugin needs to establish 4-6 connections between the two communicating applications. The plugin uses these connections to exchange DDS data (discovery or user data) and TCP Transport Plugin control messages.

With the default configuration, the TCP Transport Plugin does not support external load balancers. This is because external load balancers do not forward the traffic to a unique TCP Transport Plugin server, but they divide the connections among multiple servers. Because of this behavior, when an application running a TCP Transport Plugin client tries to establish all the connections to an application running a TCP Transport Plugin server, the server may not receive all the required connections.

In order to support external load balancers, the TCP Transport Plugin provides a session-ID negotiation feature. When session-ID negotiation is enabled (by setting the `negotiate_session_id` property to true), the TCP Transport Plugin will perform the negotiation depicted in Figure 44.1: Session-ID Negotiation on the next page.
During the session-ID negotiation, the TCP Transport Plugin exchanges three types of messages:

**Session-ID Request**: This message is sent from the client to the server. The server must respond with a session-ID response.

**Session-ID Response**: This message is sent from the server to the client as a response to a session-ID request. The client will store the session ID contained in this message.

**Session-ID Indication**: This message is sent from the client to the server; it does not require a response from the server.

The negotiation consists of the following steps:

1. The TCP client sends a session-ID request with the session ID set to zero.
2. The TCP server sends back a session-ID response with the session ID set to zero.
3. The external load balancer modifies the session-ID response, setting the session ID with a value that is meaningful to the load balancer and identifies the session.
4. The TCP client receives the session-ID response and stores the received session ID.
5. For each new connection, the TCP client sends a session-ID indication containing the stored session ID. This will allow the load balancer to redirect to the same server all the connections with the same session ID.
44.6.1 Session-ID Messages

TCP Payload for Session-ID Message below depicts the TCP payload of a session-ID message. The payload consists of 48 bytes. In particular, your load balancer needs to read/modify the following two fields:

CTRLTYPE: This field allows a load balancer to identify session-ID messages. Its value (two bytes) varies according to the session-ID message type: 0x0c05 for a request, 0x0d05 for a response, or 0x0c15 for an indication.

SESSION-ID: This field consists of 16 bytes that the load balancer can freely modify according to its requirements.

<table>
<thead>
<tr>
<th>00</th>
<th>01</th>
<th>02</th>
<th>03</th>
<th>04</th>
<th>05</th>
<th>06</th>
<th>07</th>
<th>08</th>
<th>09</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTI reserved</td>
<td>0xDD</td>
<td>0x54</td>
<td>0xDD</td>
<td>0x55</td>
<td>CTRLTYPE</td>
<td>RTI reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td>31</td>
</tr>
<tr>
<td>RTI reserved</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td>37</td>
<td>38</td>
<td>39</td>
<td>40</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
<td>45</td>
<td>46</td>
<td>47</td>
</tr>
<tr>
<td>SESSION-ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To ensure all the TCP connections within the same session are directed to the same server, you must configure your load balancer to perform the two following actions:

Modify the SESSION-ID field in the session-id response with a value that identifies the session within the load balancer.

Make the load-balancing decision according to the value of the SESSION-ID field in the session-ID indication.

44.7 TCP/TLS Transport Properties

Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t describes the TCP and TLS transport properties.

Note: To use TLS, you also need RTI TLS Support, which is a separate component.
Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.TCPv4.tcp1.')</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dds.transport.load_plugins (Note: this does not take a prefix)</td>
<td>Comma-separated strings indicating the prefix names of all plugins that will be loaded by Connext DDS. For example: &quot;dds.transport.TCPv4.tcp1&quot;. You will use this string as the prefix to the property names. Note: you can load up to 8 plugins.</td>
</tr>
<tr>
<td>library</td>
<td>Only required if linking dynamically If used, must be &quot;nddstransporttcp&quot;. This library must be in your library search path (pointed to by the environment variable LD_LIBRARY_PATH on Linux systems, Path on Windows systems, DYLD_LIBRARY_PATH on macOS systems).</td>
</tr>
<tr>
<td>create_function</td>
<td>Only required if linking dynamically If used, must be &quot;NDDS_Transport_TCPv4_create&quot;.</td>
</tr>
<tr>
<td>create_function_ptr</td>
<td>Only required if linking statically Defines the function pointer to the TCP Transport Plugin creation function. Used for loading TCP Transport Plugin statically. Must be set to the NDDS_Transport_TCPv4_create function pointer.</td>
</tr>
<tr>
<td>tls_create_function_ptr</td>
<td>Defines the function pointer to the TLS Support creation function. Used for loading TLS Support libraries statically. Must be set to the RTITLS_ConnectionEndpointFactoryTLSv4_create function pointer. Note: In order to have effect, the tls_delete_function_ptr property must also be set.</td>
</tr>
<tr>
<td>tls_delete_function_ptr</td>
<td>Defines the function pointer to the TLS Support deletion function. Used for loading TLS Support libraries statically. Must be set to the RTITLS_ConnectionEndpointFactoryTLSv4_delete function pointer. Note: In order to have effect, the tls_create_function_ptr property must also be set.</td>
</tr>
<tr>
<td>aliases</td>
<td>Used to register the transport plugin returned by NDDS_Transport_TCPv4_create() (as specified by &lt;TCP_prefix&gt;.create_function) to the DomainParticipant. Aliases should be specified as a comma-separated string, with each comma delimiting an alias. If it is not specified, the prefix--without the leading &quot;dds.transport&quot;--is used as the default alias for the plugin. For example, if the &lt;TRANSORT_PREFIX&gt; is &quot;dds.transport.mytransport&quot;, the default alias for the plugin is &quot;mytransport&quot;.</td>
</tr>
</tbody>
</table>

1Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with <code>dds.transport.TCPv4.tcp1</code>.)</th>
<th>Description</th>
</tr>
</thead>
</table>
| parent.classid                                          | Must be set to one of the following values:  
- `NDDS_TRANSPORT_CLASSID_TCPV4_LAN` for TCP communication within a LAN  
- `NDDS_TRANSPORT_CLASSID_TLSV4_LAN` for TLS communication within a LAN  
- `NDDS_TRANSPORT_CLASSID_TCPV4_WAN` for TCP communication across LANs and firewalls  
- `NDDS_TRANSPORT_CLASSID_TLSV4_WAN` for TLS communication across LAN and firewalls  
  Default: `NDDS_TRANSPORT_CLASSID_TCPV4_LAN`  
  **Note:** To use either TLS mode, you also need RTI TLS Support which is available for purchase as a separate package. |
| parent.gather_send_buffer_count_max                     | Specifies the maximum number of buffers that Connext DDS can pass to the `send()` function of the transport plugin.  
The transport plugin `send()` API supports a gather-send concept, where the `send()` call can take several discontiguous buffers, assemble and send them in a single message. This enables Connext DDS to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer.  
However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent Connext DDS from trying to gather too many buffers into a send call for the transport plugin.  
**Connext DDS** requires all transport-plugin implementations to support a gather-send of at least a minimum number of buffers. This minimum number is defined as `NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN`.  
Default: 128 |
| parent.message_size_max                                 | The maximum size of a message in bytes that can be sent or received by the transport plugin. Above this size, DDS-level fragmentation will occur. See 23.3 Large Data Fragmentation on page 970.  
Default: 65536 |

---

1 Assuming you used `dds.transport.TCPv4.tcp1` as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with `dds.transport`.
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><a href="#">parent.allow_interfaces_list</a></td>
<td>A list of strings, each identifying a range of interface addresses or an interface name that can be used by the transport. Interfaced must be specified as comma-separated strings, with each comma delimiting an interface. For example: 10.10.<em>, 10.15.</em> If the list is non-empty, this &quot;white&quot; list is applied before <a href="#">parent.deny_interfaces_list</a> below. Default: All available interfaces are used.</td>
</tr>
<tr>
<td><a href="#">parent.deny_interfaces_list</a></td>
<td>A list of strings, each identifying a range of interface addresses or an interface name that will not be used by the transport. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. For example: 10.10.* This &quot;black&quot; list is applied after <a href="#">parent.allow_interfaces_list</a> above and filters out the interfaces that should not be used. Default: No interfaces are denied</td>
</tr>
<tr>
<td>send_socket_buffer_size</td>
<td>Size, in bytes, of the send buffer of a socket used for sending. On most operating systems, setsockopt() will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to <a href="#">parent.message_size_max</a> on the previous page, or -1. The maximum value is operating system-dependent. When set to -1, setsockopt() (or equivalent) will not be called to size the send buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
<tr>
<td>recv_socket_buffer_size</td>
<td>Size, in bytes, of the receive buffer of a socket used for receiving. On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to <a href="#">parent.message_size_max</a> on the previous page, or -1. The maximum value is operating system dependent. When set to -1, setsockopt() (or equivalent) will not be called to size the receive buffer of the socket. The transport will use the OS default. Default: 131072</td>
</tr>
</tbody>
</table>

1Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ignore_loopback_interface</code></td>
<td>Prevents the transport plugin from using the IP loopback interface. This property is ignored when <code>parent.classid on page 1160</code> is NDDS_TRANSPORT_CLASSID_TCPv4_WAN or NDDS_TRANSPORT_CLASSID_TLSv4_WAN. Two values are allowed:</td>
</tr>
<tr>
<td><code>0</code></td>
<td>Enable local traffic via this plugin. The plugin will only use and report the IP loopback interface only if there are no other network interfaces (NICs) up on the system.</td>
</tr>
<tr>
<td><code>1</code></td>
<td>Disable local traffic via this plugin. This means “do not use the IP loopback interface, even if no NICs are discovered.” This setting is useful when you want applications running on the same node to use a more efficient plugin like shared memory instead of the IP loopback. Default: 1</td>
</tr>
<tr>
<td><code>ignore_nonrunning_interfaces</code></td>
<td>It prevents the transport plugin from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that “all resources are allocated” and may be off if no link is detected (e.g., the network cable is unplugged). Two values are allowed:</td>
</tr>
<tr>
<td><code>0</code></td>
<td>Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP.</td>
</tr>
<tr>
<td><code>1</code></td>
<td>Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network. Default: 1</td>
</tr>
</tbody>
</table>

---

1Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.TCPv4.tcp1.')</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>transport_priority_mask</td>
<td>Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low below/transport_priority_mapping_high below to define the mapping from DDS transport priority to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket. For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000-0xff00 in this case) to the range specified by low and high. If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets. Default: 0</td>
</tr>
<tr>
<td>transport_priority_mapping_low</td>
<td>Sets the low and high values of the output range to IPv4 TOS. These values are used in conjunction with transport_priority_mask above to define the mapping from DDS transport priority to the IPv4 TOS field. Defines the low and high values of the output range for scaling.</td>
</tr>
<tr>
<td>transport_priority_mapping_high</td>
<td>Note that IPv4 TOS is generally an 8-bit value. Default transport_priority_mapping_low: 0 Default transport_priority_mapping_high: 0xFF</td>
</tr>
<tr>
<td>interface_poll_period</td>
<td>Specifies the period in milliseconds to query for changes in the state of all the interfaces. See interface_poll_period in 16.6 Setting Built-in Transport Properties with the PropertyQosPolicy on page 801</td>
</tr>
<tr>
<td>server_socket_backlog</td>
<td>The backlog parameter determines what is the maximum length of the queue of pending connections. Default: 5</td>
</tr>
</tbody>
</table>

1 Assuming you used 'dds.transport.TCPv4.tcp1' as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| public_address | Required for WAN communication (see note below)  
Public IP address and port (WAN address and port) (separated with `:`) associated with the transport instantiation.  
For example: 10.10.9.10:4567  
This field is used only when parent.classid on page 1160 is NDDS_TRANSPORT_CLASSID_TCPV4_WAN or NDDS_TRANSPORT_CLASSID_TLSV4_WAN.  
The public address and port are necessary to support communication over WAN that involves Network Address Translators (NATs). Typically, the address is the public address of the IP router that provides access to the WAN. The port is the IP router port that is used to reach the private server_bind_port on the next page inside the LAN from the outside. This value is expressed as a string in the form: `ip[port]`, where `ip` represents the IPv4 address and port is the external port number of the router.  
Host names are not allowed in the `public_address` because they may resolve to an internet address that is not what you want (i.e., 'localhost' may map to your local IP or to 127.0.0.1).  
**Note:** If you are using an asymmetric configuration, `public_address` does not have to be set for the non-public peer. |
| bind_interface_address | The TCP transport can be configured to bind all sockets to a specified interface.  
If NULL, the sockets will be bound to the special IP address INADDR_ANY. This address allows the sockets to receive packets destined to any of the interfaces.  
This field should be set in multi-homed systems communicating across NAT routers. |

---

1 Assuming you used `dds.transport.TCPv4.tcp1` as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with `dds.transport.`
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_bind_port</td>
<td>Private IP port (inside the LAN) used by the transport to accept TCP connections. If this property is set to zero (which is only a valid configuration when parent.classid is NDDS_TRANSPORT_CLASSID_TCPV4_WAN or NDDS_TRANSPORT_CLASSID_TLSV4_WAN), the transport will operate in &quot;asymmetric mode&quot; and it will disable the internal server socket, making it impossible for external peers to connect to this node. In this case, the node is considered unreachable and will communicate only using the asymmetric mode with other (reachable) peers. For more information about the available modes of operation for the transport, please refer to Chapter 43 TCP Communication Scenarios on page 1143. For WAN communication, if server_bind_port is set to a value other than zero, this port must be forwarded to a public port in the NAT-enabled router that connects to the outer network. The server_bind_port cannot be shared among multiple participants on a common host. On most operating systems, attempting to reuse the same server_bind_port for multiple participants on a common host will result in a &quot;port already in use&quot; error. However, Windows systems will not recognize if the server_bind_port is already in use; therefore care must be taken to properly configure Windows systems. Default: 7400</td>
</tr>
</tbody>
</table>
| read_buffer_allocation        | Allocation settings applied to read buffers. These settings configure the initial number of buffers, the maximum number of buffers and the buffers to be allocated when more buffers are needed. Default:  
  - read_buffer_allocation.initial_count = 2  
  - read_buffer_allocation.max_count = -1 (unlimited)  
  - read_buffer_allocation.incremental_count = -1 (number of buffers will keep doubling on each allocation until it reaches max_count) |

---

1Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ’dds.transport.’
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
</table>
| **write_buffer_allocation**       | Allocation settings applied to buffers used for asynchronous (non-blocking) send. To enable asynchronous send, set the property `force_asynchronous_send` to 1. These settings configure the initial number of buffers, the maximum number of buffers, and the buffers to be allocated when more buffers are needed. Default:  

  - `write_buffer_allocation.initial_count = 4`
  - `write_buffer_allocation.max_count = 32`
  - `write_buffer_allocation.incremental_count = 2`

  The pool of buffers can be configured to be shared across all the TCP connections created by the TCP Transport or to be exclusive for a connection by setting the property `shared_write_buffer_allocation` to 1. The default value is 0. Note that for the write buffer pool, the `max_count` is not set to unlimited. This is to avoid having a fast writer quickly exhaust all the available system memory, in case of a temporary network slowdown. When this write buffer pool reaches the maximum, a new message will replace the oldest message that is not currently in the process of being sent. This guarantees that new messages are prioritized, while at the same time not running into a situation in which messages are not received. Messages that are replaced and not sent may be resent later depending on the application’s QoS (if the transport is used for reliable communication, the data will still be sent eventually). |
| **shared_write_buffer_allocation** | This property determines whether the pool of buffers created with asynchronous (non-blocking) send is shared or exclusive per TCP connection. Sharing this buffer across connections may lead to less memory consumption. However, high-throughput connections may starve low-throughput connections. This is why the default value is 0. The size of the buffer pool can be configured using the property `write_buffer_allocation`. Default: 0 |
| **control_buffer_allocation**     | Allocation settings applied to buffers used to serialize and send control messages. These settings configure the initial number of buffers, the maximum number of buffers and the buffers to be allocated when more buffers are needed. Default:  

  - `control_buffer_allocation.initial_count = 2`
  - `control_buffer_allocation.max_count = -1` (unlimited)
  - `control_buffer_allocation.incremental_count = -1` (number of buffers will keep doubling on each allocation until it reaches `max_count`)  |

---

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.TCPv4.tcp1.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>control_message_allocation</td>
<td>Allocation settings applied to control messages. These settings configure the initial number of messages, the maximum number of messages and the messages to be allocated when more messages are needed. Default:</td>
</tr>
<tr>
<td></td>
<td>- control_message_allocation.initial_count = 2</td>
</tr>
<tr>
<td></td>
<td>- control_message_allocation.max_count = -1 (unlimited)</td>
</tr>
<tr>
<td></td>
<td>- control_message_allocation.incremental_count = -1 (number of messages will keep doubling on each allocation until it reaches max_count)</td>
</tr>
<tr>
<td>control_attribute_allocation</td>
<td>Allocation settings applied to control messages attributes. These settings configure the initial number of attributes, the maximum number of attributes and the attributes to be allocated when more attributes are needed. Default:</td>
</tr>
<tr>
<td></td>
<td>- control_attribute_allocation.initial_count = 2</td>
</tr>
<tr>
<td></td>
<td>- control_attribute_allocation.max_count = -1 (unlimited)</td>
</tr>
<tr>
<td></td>
<td>- control_attribute_allocation.incremental_count = -1 (number of attributes will keep doubling on each allocation until it reaches max_count)</td>
</tr>
<tr>
<td>force_asynchronous_send</td>
<td>Forces asynchronous send. When this parameter is set to 0, the TCP Transport will attempt to send data as soon as the internal send() function is called. When it is set to 1, the transport will make a copy of the data to send in an internal send buffer and enqueue it. Data will be sent as soon as the low-level socket buffer has space. Setting this option to 0 (default) should provide better latency. However, in high-throughput scenarios, a 0 setting may cause the low-level send() function to block until the data is physically delivered to the lower socket buffer. For an application writing data at a very fast rate, the 0 setting may cause the caller thread to block if the send socket buffer is full. This could produce lower throughput in those conditions (the caller thread could prepare the next packet while waiting for the send socket buffer to become available). The size of the buffer pool created by setting this option to 1 can be configured using the property write_buffer_allocation. In addition, the TCP Transport can be used to create one buffer pool per connection or a single buffer pool shared across all TCP connections by using the property shared_write_buffer_allocation. Default: 0</td>
</tr>
</tbody>
</table>

---

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'

1167
Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.TCPv4.tcp1.')</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_packet_size</td>
<td>The maximum size of a TCP segment. This parameter is only supported on Linux architectures. By default, the maximum size of a TCP segment is based on the network MTU for destinations on a local network, or on a default 576 for destinations on non-local networks. This behavior can be changed by setting this parameter to a value between 1 and 65535. Default: -1 (default behavior)</td>
</tr>
<tr>
<td>enable_keep_alive</td>
<td>Configures the sending of KEEP_ALIVE messages in TCP. Setting this value to 1, causes a KEEP_ALIVE packet to be sent to the remote peer if a long time passes with no other data sent or received. This feature is implemented only on architectures that provide a low-level implementation of the TCP keep-alive feature. On Windows systems, the TCP keep-alive feature can be globally enabled through the system’s registry: \HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Tcpip\Parameters. Refer to MSDN documentation for more details. Default: 0</td>
</tr>
<tr>
<td>keep_alive_time</td>
<td>Specifies the interval of inactivity in seconds that causes TCP to generate a KEEP_ALIVE message. This parameter is only supported on Linux and Mac architectures. Default: -1 (OS default value)</td>
</tr>
<tr>
<td>keep_alive_interval</td>
<td>Specifies the interval in seconds between KEEP_ALIVE retries. This parameter is only supported on Linux architectures. Default: -1 (OS default value)</td>
</tr>
<tr>
<td>keep_alive_retry_count</td>
<td>The maximum number of KEEP_ALIVE retries before dropping the connection. This parameter is only supported on Linux architectures. Default: -1 (OS default value)</td>
</tr>
</tbody>
</table>

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ‘dds.transport.’
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.TCPv4.tcp1.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>user_timeout</td>
<td>Changes the default OS TCP User Timeout configuration. If set to a value greater than 0, it specifies the maximum amount of time in seconds that transmitted data may remain unacknowledged before TCP will forcibly close the corresponding connection and return ETIMEDOUT to the application. If set to 0, TCP Transport plugin will use the system default. Currently this feature is supported only on Linux 2.6.37 and higher platforms. Default: 0 (use system's default).</td>
</tr>
<tr>
<td>connection_liveliness</td>
<td>Configures the connection liveliness feature. See 44.7.1 Connection Liveliness on page 1177. Defaults:</td>
</tr>
<tr>
<td></td>
<td>- connection_liveliness.enable: 0</td>
</tr>
<tr>
<td></td>
<td>- connection_liveliness.lease_duration: 10</td>
</tr>
<tr>
<td></td>
<td>- connection_liveliness.assertions_per_lease_duration: 3</td>
</tr>
<tr>
<td>event_thread</td>
<td>Configures the event thread used by the TCP Transport plugin for providing some features. Defaults:</td>
</tr>
<tr>
<td></td>
<td>- event_thread.priority: THREAD_PRIORITY_DEFAULT</td>
</tr>
<tr>
<td></td>
<td>- event_thread.stack_size: THREAD_STACK_SIZE_DEFAULT</td>
</tr>
<tr>
<td></td>
<td>- event_thread.mask: PRIORITY_ENFORCE</td>
</tr>
<tr>
<td>disable_nagle</td>
<td>Disables the TCP nagle algorithm. When this property is set to 1, TCP segments are always sent as soon as possible, which may result in poor network utilization. Default: 0</td>
</tr>
</tbody>
</table>

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.TCPv4.tcp1.’)</th>
<th>Description</th>
</tr>
</thead>
</table>
| logging_verbosity_bitmap | Bitmap that specifies the verbosity of log messages from the transport. Logging values:  
- -1 (0xffffffff): do not change the current verbosity  
- 0x00: silence  
- 0x01: fatal error  
- 0x02: errors  
- 0x04: warnings  
- 0x08: local  
- 0xb0: remote  
- 0xb2: periodic  
- 0xb0: other (used for control protocol tracing)  
- 13F: all (fatal error, errors, warnings, local, remote, periodic, and other)  
  You can combine these values by logically ORing them together.  
  Default: -1 (meaning, do not change the current verbosity, which is fatal errors, errors, and warnings)  
  Note: the logging verbosity is a global property shared across multiple instances of the TCP Transport. If you create a new TCP Transport instance with logging_verbosity_bitmap different than -1, the change will affect all the other instances as well.  
  Note: The option of 0x100 (other) is used only for tracing the internal control protocol. Since the output is very verbose, this feature is enabled only in the debug version of the TCP Transport library ([libnddstransporttcpd.so](https://libddstransporttcpd.so/) / [LIBNDDSTRANSPORTD.LIB](https://libddstransporttcpd.so/)). |

---

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ‘dds.transport.’
Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.TCPv4.tcp1.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property_name</td>
<td>Bitmap that specifies the verbosity of security related log messages from the transport. These are usually messages generated by OpenSSL. Logging values:</td>
</tr>
<tr>
<td></td>
<td>• <code>-1 (0xffffffff): inherit logging_verbosity_bitmap value</code></td>
</tr>
<tr>
<td></td>
<td>• <code>0x00: silence</code></td>
</tr>
<tr>
<td></td>
<td>• <code>0x01: fatal error</code></td>
</tr>
<tr>
<td></td>
<td>• <code>0x02: errors</code></td>
</tr>
<tr>
<td></td>
<td>• <code>0x04: warnings</code></td>
</tr>
<tr>
<td></td>
<td>• <code>0x08: local</code></td>
</tr>
<tr>
<td></td>
<td>• <code>0x10: remote</code></td>
</tr>
<tr>
<td></td>
<td>• <code>0x20: periodic</code></td>
</tr>
<tr>
<td>You can combine these values by logically ORing them together.</td>
<td></td>
</tr>
<tr>
<td>Default: <code>-1 (inherit logging_verbosity_bitmap value)</code></td>
<td></td>
</tr>
<tr>
<td>Note: The security logging verbosity is a global property shared across multiple instances of the TCP Transport. If you create a new TCP Transport instance with a <code>security_verbosity_bitmap</code> other than <code>-1</code>, the change will affect all the other instances as well.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>socket_monitoring_kind</td>
<td>Configures the socket monitoring API used by the transport. This property can have the following values:</td>
</tr>
<tr>
<td></td>
<td>• <code>SELECT</code>: The transport uses the POSIX select API to monitor sockets.</td>
</tr>
<tr>
<td></td>
<td>• <code>WINDOWS_IOCP</code>: The transport uses Windows I/O completion ports to monitor sockets. This value only applies to Windows systems.</td>
</tr>
<tr>
<td></td>
<td>• <code>WINDOWS_WAITFORMULTIPLEOBJECTS</code>: The transport uses the API <code>WaitForMultipleObjects</code> to monitor sockets. This value only applies to Windows systems.</td>
</tr>
<tr>
<td>Default: <code>SELECT</code></td>
<td></td>
</tr>
<tr>
<td>Note: The value selected for this property may affect transport performance and scalability. On Windows systems, using <code>WINDOWS_IOCP</code> provides the best performance and scalability.</td>
<td></td>
</tr>
</tbody>
</table>

---

1Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ‘dds.transport.’
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.TCPv4.tcp1.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>windows_iocp</td>
<td>Configures I/O completion ports when socket_monitoring_kind on the previous page is set to WINDOWS_IOP. This setting configures the number of threads the plugin creates to process I/O completion packets (thread_pool_size) and the number of those threads that the operating system can allow to concurrently run (concurrency_value). Defaults: windows_iocp.thread_pool_size: 2 windows_iocp.concurrency_value: 1</td>
</tr>
<tr>
<td>send_crc</td>
<td>When set to 1, enables the computation of the CRC for sent RTI TCP messages. Default: 0</td>
</tr>
<tr>
<td>force_crc_check</td>
<td>When set to 1, forces the checking of the CRC for received RTI TCP messages. By default, the TCP Transport plugin will only validate the CRC if the CRC is present in the received message. If this property is set to 1, TCP Transport will drop messages not including the CRC. Default: 0</td>
</tr>
<tr>
<td>negotiate_session_id</td>
<td>When set to 1, the TCP Transport Plugin will perform a session negotiation that will help external load balancers identify all the connections associated with a particular session between two Connext DDS applications. This keeps the connections from being divided among multiple servers and ensures proper communication. For more information about this property, see 44.6 Support for External Hardware Load Balancers in TCP Transport Plugin on page 1156. Default: 0 <strong>Note:</strong> The value of this property must be consistent among all the applications running the TCP Transport Plugin. If two applications have a different value for this property, they may not communicate.</td>
</tr>
<tr>
<td>outstanding_connection_cookies</td>
<td>Maximum number of outstanding connection cookies allowed by the transport when acting as server. A connection cookie is a token provided by a server to a client; it is used to establish a data connection. Until the data connection is established, the cookie cannot be reused by the server. To avoid wasting memory, it is good practice to set a cap to the maximum number of connection cookies (pending connections). When the maximum value is reached, a client will not be able to connect to the server until new cookies become available. Range: 1 or higher, or -1 (which means an unlimited number). Default: 100</td>
</tr>
</tbody>
</table>

1Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>outstanding_connection_cookies_life_span</code></td>
<td>Maximum lifespan (in seconds) of the cookies associated with pending connections. If a client does not connect to the server before the lifespan of its cookie expires, it will have to request a new cookie. Range: 1 second or higher, or -1. Default: -1, which means an unlimited amount of time (effectively disabling the feature).</td>
</tr>
<tr>
<td><code>send_max_wait_sec</code></td>
<td>Controls the maximum time (in seconds) the low-level <code>sendto()</code> function is allowed to block the caller thread when the TCP send buffer becomes full. If the bandwidth used by the transport is limited, and the sender thread tries to push data faster than the OS can handle, the low-level <code>sendto()</code> function will block the caller until there is some room available in the queue. Limiting this delay eliminates the possibility of deadlock and increases the response time of the internal DDS thread. This property affects both CONTROL and DATA streams. It only affects SYNCHRONOUS send operations. Asynchronous sends never block a send operation. For synchronous <code>send()</code> calls, this property limits the time the DDS sender thread can block for a full send buffer. If it is set too large, Connext DDS not only won’t be able to send more data, it also won’t be able to receive any more data because of an internal resource mutex. Setting this property to 0 causes the low-level function to report an immediate failure if the TCP send buffer is full. Setting this property to -1 causes the low-level function to block forever until space becomes available in the TCP buffer. Default: 3 seconds.</td>
</tr>
<tr>
<td><code>client_connection_negotiation_timeout</code></td>
<td>Timeout (in seconds) for negotiating a client data connection. The TCP Transport plugin requires some negotiation before establishing a connection. This property controls the maximum time (in seconds) a client data connection negotiation can remain in progress. In particular, it controls a maximum timeout for requesting and replying to a server logical port request. If the negotiation of a connection has not completed after the specified timeout, the negotiation will restart, and if there is an associated data connection, it will be closed. This way, the TCP Transport plugin can retry the process of establishing and negotiating that connection. Range: 1 second or higher. Default: 10 seconds</td>
</tr>
</tbody>
</table>

---

1 Assuming you used `dds.transport.TCPv4.tcp1` as the alias to load the plugin. If not, change the prefix to match the string used with `dds.transport.load_plugins`. This prefix must begin with `dds.transport.`
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.TCPv4.tcp1.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>server_connection_negotiation_timeout</td>
<td>Timeout (in seconds) for negotiating a server data connection. The TCP Transport plugin requires some negotiation before establishing a connection. This property controls the maximum time (in seconds) a server data connection negotiation can remain in progress. In particular, it controls a maximum timeout for requesting and replying to a client logical port request. If the negotiation of a connection has not completed after the specified timeout, the negotiation will restart, and if there is an associated data connection, it will be closed. This way, the TCP Transport plugin can retry the process of establishing and negotiating that connection. Range: 1 second or higher. Default: 10 seconds</td>
</tr>
<tr>
<td>initial_handshake_timeout</td>
<td>Timeout (in seconds) for the initial handshake for a connection. Once a connection is established, TCP transport will exchange some information to identify itself and the connection. This process is known as the initial handshake of a connection, and if using TLS the TCP Transport plugin will also exchange additional information to secure the connection. This property controls the maximum time (in seconds) the initial handshake for a connection can remain in progress. If the handshake has not completed after the specified timeout, the connection will be closed. This way, the TCP Transport plugin can restart the process of establishing and handshaking that connection. Range: 1 second or higher. Default: 10 seconds</td>
</tr>
<tr>
<td>tls.verify.ca_file</td>
<td>A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for SSL_load_verify_locations for more information. <strong>To enable TLS, ca_file or ca_path is required; both may be specified (at least one is required).</strong></td>
</tr>
<tr>
<td>tls.verify.ca_path</td>
<td>A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for SSL_CTX_load_verify_locations for more information. The Certificate Authority subject name hash values must be available in the directories. You may generate them by running openssl rehash (available in OpenSSL 1.1.0 or above) or <em>c_rehash</em> in each directory. <strong>To enable TLS, ca_file or ca_path is required; both may be specified (at least one is required).</strong></td>
</tr>
</tbody>
</table>

---

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with ‘dds.transport.TCPv4.tcp1.’)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls.verify.verify_depth</td>
<td>Maximum certificate chain length for verification.</td>
</tr>
<tr>
<td>tls.verify.crl_file</td>
<td>Name of the file containing the Certificate Revocation List. File should be in PEM format.</td>
</tr>
<tr>
<td>tls.identity.certificate_chain</td>
<td>An identifying certificate is required for secure communication. The string must be sorted starting with the certificate to the highest level (root CA). If this is specified, certificate_chain_file must be empty.</td>
</tr>
<tr>
<td>tls.identity.certificate_chain_file</td>
<td>File containing identifying certificate (in PEM format) or certificate chain (appending intermediate CA certs in order). An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If this is specified, certificate_chain_file must be empty. Optionally, a private key may be appended to this file. If no private key option is specified, this file will be used to load a private key.</td>
</tr>
<tr>
<td>tls.identity.private_key_password</td>
<td>A string that specifies the password for private key.</td>
</tr>
<tr>
<td>tls.identity.private_key</td>
<td>String containing private key (in PEM format). At most one of private_key and private_key_file may be specified. If no private key is specified (all values are NULL), the private key will be read from the certificate chain file.</td>
</tr>
<tr>
<td>tls.identity.private_key_file</td>
<td>File containing private key (in PEM format). At most one of private_key and private_key_file may be specified. If no private key is specified (all values are NULL), the private key will be read from the certificate chain file.</td>
</tr>
<tr>
<td>tls.identity.rsa_private_key</td>
<td>String containing additional RSA private key (in PEM format). For use if both an RSA and non-RSA key are required for the selected cipher. At most one of rsa_private_key and rsa_private_key_file may be specified.</td>
</tr>
<tr>
<td>tls.identity.rsa_private_key_file</td>
<td>File containing additional RSA private key (in PEM format). For use if both an RSA and non-RSA key are required for the selected cipher. At most one of rsa_private_key and rsa_private_key_file may be specified.</td>
</tr>
</tbody>
</table>

---

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ‘dds.transport.’
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tls.cipher.cipher_list</td>
<td>List of available TLS ciphers when communicating with Connext DDS 6.0.0 or below. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string. Default: NULL</td>
</tr>
<tr>
<td>tls.cipher.ciphersuites</td>
<td>List of available TLS ciphersuites when communicating with Connext DDS 6.0.1 or above. See the OpenSSL manual page for SSL_CTX_set_ciphersuites for more information on the format of this string. Default: NULL</td>
</tr>
<tr>
<td>tls.cipher.dh_param_files</td>
<td>List of available Diffie-Hellman (DH) key files. For example: “foo.h:2048,bar.h:1024” means: dh_param_files[0].file = foo.pem, dh_param_files[0].bits = 2048, dh_param_files[1].file = bar.pem, dh_param_files[1].bits = 1024</td>
</tr>
<tr>
<td>tls.cipher.engine_id</td>
<td>ID of OpenSSL cipher engine to request.</td>
</tr>
<tr>
<td>disable_interface_tracking</td>
<td>If this variable is set, the automatic change detection over the system network interfaces will be disabled. See disable_interface_tracking in 16.6 Setting Built-in Transport Properties with the PropertyQosPolicy on page 801</td>
</tr>
<tr>
<td>force_interface_poll_detection</td>
<td>This property forces the interface tracker to use a polling method to detect changes to the network interfaces in IP mobility scenarios. It only applies to operating systems that support asynchronous notifications of interface changes. If set to TRUE, the interface tracker will use a polling method that queries the interfaces periodically to detect the changes. If set to FALSE, the interface tracker will use the operating system’s default method. Basically, this property allows you—for an operating system that supports asynchronous notification—to use the polling method instead. Default: FALSE</td>
</tr>
</tbody>
</table>

---

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with ’dds.transport.’
### Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t

<table>
<thead>
<tr>
<th>Property Name (prefix with 'dds.transport.TCPv4.tcp1.')</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>property_validation_action</td>
<td>By default, property names given in the 7.5.19 PROPERTY QosPolicy (DDS Extension) on page 439 are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the transport:</td>
</tr>
<tr>
<td></td>
<td>- VALIDATION_ACTION: validate the properties. Upon failure, log errors and fail.</td>
</tr>
<tr>
<td></td>
<td>- VALIDATION_ACTION_SKIP: skip validation.</td>
</tr>
<tr>
<td></td>
<td>- VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail.</td>
</tr>
<tr>
<td></td>
<td>If this property is not set, the property validation behavior will be the same as that of the DomainParticipant, which by default is VALIDATION_ACTION_EXCEPTION. See 7.5.19.1 Property Validation on page 441 for more information.</td>
</tr>
<tr>
<td>thread_name_prefix</td>
<td>You can set this field with your own value, to help you identify the transport thread in a way that's meaningful to you. Do not exceed 8 characters.</td>
</tr>
<tr>
<td></td>
<td>If you do not set this field, Connext DDS creates the following prefix:</td>
</tr>
<tr>
<td></td>
<td>'r' + 'Tr' + participant identifier + '\0'</td>
</tr>
<tr>
<td></td>
<td>Where 'r' indicates this is a thread from RTI, 'Tr' indicates the thread is related to a transport, and participant identifier contains 5 characters as follows:</td>
</tr>
<tr>
<td></td>
<td>- If participant_name is set: The participant identifier will be the first 3 characters and the last 2 characters of the participant_name.</td>
</tr>
<tr>
<td></td>
<td>- If participant_name is not set, then the identifier is computed as domain_id (3 characters) followed by participant_id (2 characters).</td>
</tr>
<tr>
<td></td>
<td>- If participant_name is not set and the participant_id is set to -1 (default value), then the participant identifier is computed as the last 5 digits of the rtps_instance_id in the participant GUID.</td>
</tr>
</tbody>
</table>

See 21.9 Identifying Threads Used by Connext DDS on page 916.

### 44.7.1 Connection Liveliness

The connection_liveliness property configures the connection liveliness feature. When enabled, the TCP Transport plugin will periodically exchange some additional control traffic (liveliness requests/responses) over one of the connections between the TCP Client and Server. This traffic allows determining if a that connection is not alive anymore, and thus proceed to its close. This avoids depending on the OS notification about the status of the connection, potentially decreasing the time to reestablish lost connections.

1 Assuming you used ‘dds.transport.TCPv4.tcp1’ as the alias to load the plugin. If not, change the prefix to match the string used with dds.transport.load_plugins. This prefix must begin with 'dds.transport.'
The following parameters can be configured:

- **connection_liveliness.enable**: Enables or disables the feature.
- **connection_liveliness.lease_duration**: In seconds, the timeout by which the connection liveliness must be asserted or the connection will be considered not alive. It is also used as the period between connection liveliness checks. Therefore, the maximum time before a connection is marked as not alive is $2 \times \text{connection_liveliness.lease_duration}$.
- **connection_liveliness.assertions_per_lease_duration**: The number of liveliness requests sent per each lease duration. Increasing this value will increase the overhead sent into the network, but it will also make the connection liveliness mechanism more robust.

This feature relies on the creation of an additional thread in the TCP Transport Plugin (the event thread). For more information about how to configure this thread, see the `event_thread` in Table 44.1 Properties for NDDS_Transport_TCPv4_Property_t.

Enabling this feature breaks backwards compatibility with TCP Transport plugins that do not include this feature.
Part 8: RTI Persistence Service

The material in this part of the manual describes *Persistence Service*. It saves DDS data samples so they can be delivered to subscribing applications that join the system at a later time—even if the publishing application has already terminated.

*Persistence Service* is not available on all platforms. See the [RTI Connext DDS Core Libraries Platform Notes](#).

This section includes:

- Introduction to RTI Persistence Service (Chapter 45 on page 1180)
- Configuring Persistence Service (Chapter 46 on page 1181)
- Running RTI Persistence Service (Chapter 47 on page 1207)
- Administering Persistence Service from a Remote Location (Chapter 48 on page 1211)
- Advanced Persistence Service Scenarios (Chapter 49 on page 1217)
Chapter 45 Introduction to RTI Persistence Service

Persistence Service is a Connext DDS application that saves DDS data samples to transient or permanent storage, so they can be delivered to subscribing applications that join the system at a later time—even if the publishing application has already terminated.

Persistence Service runs as a separate application; you can run it on the same node as the publishing application, the subscribing application, or some other node in the network.

When configured to run in PERSISTENT mode (<persistent_storage> is used), Persistence Service can use the filesystem or a relational database that provides an ODBC driver. For each persistent topic, it collects all the data written by the corresponding persistent DataWriters and stores them into persistent storage. See the RTI Persistence Service Release Notes for the list of platforms and relational databases that have been tested.

When configured to run in TRANSIENT mode (<persistent_storage> is not used), Persistence Service stores the data in memory.

The following chapters assume you have a basic understanding of DDS terms such as DomainParticipants, Publishers, DataWriters, Topics, and Quality of Service (QoS) policies. For an overview of DDS terms, please see Data-Centric Publish-Subscribe Communications (Chapter 2 on page 14). You should also have already read Mechanisms for Achieving Information Durability and Persistence (Chapter 13 on page 732).
Chapter 46 Configuring Persistence Service

To use Persistence Service:

1. Modify your Connext DDS applications.
   - The 7.5.9 DURABILITY QosPolicy on page 411 controls whether or not, and how, published DDS samples are stored by Persistence Service for delivery to late-joining DataReaders. See 13.5 Data Durability on page 749.
     - For each DataWriter whose data must be stored, set the Durability QosPolicy’s kind to DDS_PERSISTENT_DURABILITY_QOS or DDS_TRANSIENT_DURABILITY_QOS.
     - For each DataReader that needs to receive stored data, set the Durability QosPolicy’s kind to DDS_PERSISTENT_DURABILITY_QOS or DDS_TRANSIENT_DURABILITY_QOS.
   - Optionally, modify the 7.5.10 DURABILITY SERVICE QosPolicy on page 416, which can be used to configure Persistence Service.

By default, the History and ResourceLimits QosPolicies for a Persistence Service DataReader (PRSTDataReader) and Persistence Service DataWriter (PRSTDataWriter) with topic 'A' will be configured using the values specified in the XML file (unless you use the tag <use_durability_service> in the persistence group definition, see 46.8 Creating Persistence Groups on page 1193). Setting the <use_durability_service> tag to true will cause the History and ResourceLimits QosPolicies for a PRSTDataReader and PRSTDataWriter to be configured using the 7.5.10 DURABILITY SERVICE QosPolicy on page 416 of the first-discovered DataWriter publishing 'A'. (For more information on the PRSTDataReader and PRSTDataWriter, see 13.5.1 RTI Persistence Service on page 749.)

2. Create a configuration file or edit an existing file, as described in 46.2 XML Configuration File on page 1183.
3. Start Persistence Service with your configuration file, as described in 47.1 Starting Persistence Service on page 1207. You can start it on either application’s node, or even an entirely different node (provided that node is included in one of the applications’ NDDS_DISCOVERY_PEERS lists).

### 46.1 How to Load the Persistence Service XML Configuration

Persistence Service loads its XML configuration from multiple locations. This section presents the various approaches, listed in load order.

The first three locations only contain QoS Profiles and are inherited from Connext DDS (see Configuring QoS with XML (Chapter 19 on page 848)).

- **$NDDSHOME/resource/xml/NDDS_QOS_PROFILES.xml**
  
  This file contains the DDS default QoS values; it is loaded automatically if it exists. *(First to be loaded.)*

- **File specified in the NDDS_QOS_PROFILES Environment Variable**

  The files (or XML strings) separated by semicolons referenced in this environment variable are loaded automatically.

- **<working directory>/USER_QOS_PROFILES.xml**

  This file is loaded automatically if it exists.

The next locations are specific to Persistence Service.

- **<NDDSHOME>/resource/xml/RTI_PERSISTENCE_SERVICE.xml**

  This file contains the default Persistence Service configurations; it is loaded if it exists. There are two default configurations: *default* and *defaultDisk*. The *default* configuration persists all the topics into memory. The *defaultDisk* configuration persists all the topics into files located in the current working directory.

- **<working directory>/USER_PERSISTENCE_SERVICE.xml**

  This file is loaded automatically if it exists.

- **File specified using the command line option, -cfgFile**

  The command-line option **-cfgFile** (see Table 47.1 Persistence Service Command-Line Options) can be used to specify a configuration file.
46.2 XML Configuration File

The configuration file uses XML format. Let's look at a very basic configuration file, just to get an idea of its contents. You will learn the meaning of each line as you read the rest of this section.

Example Configuration File

```xml
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A Configuration file may be used by several persistence services specifying multiple <persistence_service> entries -->
<dds>
    <!-- QoS LIBRARY SECTION -->
    <qos_library name="QosLib1">
        <qos_profile name="QosProfile1">
            <datawriter_qos name="WriterQos1">
                <history>
                    <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
                </history>
            </datawriter_qos>
            <datareader_qos name="ReaderQos1">
                <reliability>
                    <kind>DDS_RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
                <history>
                    <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
                </history>
            </datareader_qos>
        </qos_profile>
    </qos_library>
    <!-- PERSISTENCE SERVICE SECTION -->
    <persistence_service name="Srv1">
        <!-- REMOTE ADMINISTRATION SECTION -->
        <administration>
            <domain_id>72</domain_id>
            <distributed_logger>
                <enabled>true</enabled>
            </distributed_logger>
        </administration>
        <!-- PERSISTENT STORAGE SECTION -->
        <persistent_storage>
            <filesystem>
                <directory>/tmp</directory>
                <file_prefix>PS</file_prefix>
            </filesystem>
        </persistent_storage>
        <!-- DOMAINPARTICIPANT SECTION -->
        <participant name="Part1">
            <domain_id>71</domain_id>
            <!-- PERSISTENCE GROUP SECTION -->
            <persistence_group name="PerGroup1" filter="*">
                <single_publisher>true</single_publisher>
                <single_subscriber>true</single_subscriber>
                <datawriter_qos base_name="QosLib1::QosProfile1"/>
                <datareader_qos base_name="QosLib1::QosProfile1"/>
            </persistence_group>
        </participant>
    </persistence_service>
</dds>
```
46.2.1 Configuration File Syntax

The configuration file must follow these syntax rules:

- The syntax is XML and the character encoding is UTF-8.
- Opening tags are enclosed in <>; closing tags are enclosed in </>.
- A value is a UTF-8 encoded string. Legal values are alphanumeric characters. All leading and trailing spaces are removed from the string before it is processed.

For example, "<tag> value </tag>" is the same as "<tag>value</tag>".
- All values are case-sensitive unless otherwise stated.
- Comments are enclosed as follows: <!-- comment -->.
- The root tag of the configuration file must be <dds> and end with </dds>.
- The primitive types for tag values are specified in Table 46.1 Supported Tag Values.

<table>
<thead>
<tr>
<th>Type</th>
<th>Format</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDS_Boolean</td>
<td>yes, 1, true, BOOLEAN_TRUE or DDS_BOOLEAN_TRUE: these all mean TRUE</td>
<td>Not case-sensitive</td>
</tr>
<tr>
<td></td>
<td>no, 0, false, BOOLEAN_FALSE or DDS_BOOLEAN_FALSE: these all mean FALSE</td>
<td></td>
</tr>
<tr>
<td>DDS_Enum</td>
<td>A string. Legal values are those listed in the C or Java API Reference HTML documentation.</td>
<td>Must be specified as a string. (Do not use numeric values.)</td>
</tr>
<tr>
<td>DDS_Long</td>
<td>-2147483648 to 2147483647 or 0x80000000 to 0x7fffffff or LENGTH_UNLIMITED or DDS_LENGTH_UNLIMITED</td>
<td>A 32-bit signed integer</td>
</tr>
<tr>
<td>DDSUnsignedLong</td>
<td>0 to 4294967296 or 0 to 0xffffffff</td>
<td>A 32-bit unsigned integer</td>
</tr>
<tr>
<td>String</td>
<td>UTF-8 character string</td>
<td>All leading and trailing spaces are ignored between two tags</td>
</tr>
</tbody>
</table>
46.2.2 XML Validation

46.2.2.1 Validation at Run Time

Persistence Service validates the input XML files using a built-in Document Type Definition (DTD). You can find a copy of the built-in DTD in `<NDDSHOME>/resource/schema/rti_persistence_service.dtd`. (This is only a copy of what the Persistence Service core uses. Changing this file has no effect unless you specify its path with the DOCTYPE tag, described below.)

You can overwrite the built-in DTD by using the XML tag, `<!DOCTYPE>`. For example, the following indicates that Persistence Service must use a different DTD file to perform validation:

```xml
<!DOCTYPE dds SYSTEM
  "http:/local/usr/rti/dds/modified_rtiperserviceservice.dtd">
```

If you do not specify the DOCTYPE tag in the XML file, the built-in DTD is used.

The DTD path can be absolute, or relative to the application’s current working directory.

46.2.2.2 Validation During Editing

Persistence Service provides DTD and XSD files that describe the format of the XML content. We recommend including a reference to one of these documents in the XML file that contains the persistence service’s configuration—this provides helpful features in code editors such as Visual Studio and Eclipse, including validation and auto-completion while you are editing the XML file. Including a reference to the XSD file in the XML documents provides stricter validation and better auto-completion than the corresponding DTD file.

The DTD and XSD definitions of the XML elements are in `<NDDSHOME>/resource/schema/rti_persistence_service.dtd` and `rti_persistence_service.xsd`, respectively.

To include a reference to the XSD document in your XML file, use the attribute `xs:noNamespaceSchemaLocation` in the `<dds>` tag. For example (in the following, replace `<NDDSHOME>` with the Connext DDS installation directory, see Paths Mentioned in Documentation on page 1):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="<NDDSHOME>/resource/schema/rti_persistence_service.xsd">
  ...
</dds>
```

To include a reference to the DTD document in your XML file, use the `<!DOCTYPE>` tag. For example (in the following, replace `<NDDSHOME>` with the Connext DDS installation directory):

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dds SYSTEM
  "<NDDSHOME>/resource/schema/rti_persistence_service.dtd">
```

See Paths Mentioned in Documentation on page 1.
46.3 QoS Configuration

Each persistence group and participant has a set of DDS QoSs. There are six tags:

- `<domain_participant_qos>`
- `<publisher_qos>`
- `<subscriber_qos>`
- `<topic_qos>`
- `<datawriter_qos>`
- `<datareader_qos>`

Each QoS is identified by a name. The QoS can inherit its values from other QoSs described in the XML file. For example:

```xml
<datawriter_qos name="DerivedWriterQos" base_name="Lib::BaseWriterQos">
  <history>
    <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
  </history>
</datawriter_qos>
```

In the above example, the writer QoS named 'DerivedWriterQos' inherits the values from the writer QoS 'BaseWriterQos' contained in the library 'Lib'. The HistoryQosPolicy `kind` is set to DDS_KEEP_ALL_HISTORY_QOS.

Each XML tag with an associated name can be uniquely identified by its fully qualified name in C++ style. For more information on tags, see Configuring QoS with XML (Chapter 19 on page 848)

The persistence groups and participants can use QoS libraries and profiles to configure their QoS values. For example:

```xml
<dds>
  <!-- QoS LIBRARY SECTION -->
  <qos_library name="QosLib1">
    <qos_profile name="QosProfile1">
      <datawriter_qos name="WriterQos1">
        <history>
          <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
        </history>
      </datawriter_qos>
    </qos_profile>
  </qos_library>
  <!-- PERSISTENCE SERVICE SECTION -->
  <persistence_service name="Srv1">
    ...
  </persistence_service>
  <!-- PERSISTENCE GROUP SECTION -->
</dds>
```
46.4 Configuring the Persistence Service Application

Each execution of the Persistence Service application is configured using the content of a tag: `<persistence_service>`. When you start Persistence Service (described in 47.1 Starting Persistence Service on page 1207), you must specify which `<persistence_service>` tag to use to configure the service.

For example:

```xml
<dds>
  <persistence_service name="Srv1">
    ...
  </persistence_service>
</dds>
```

If you do not specify a service name when you start Persistence Service, the service will print the list of available configurations and then exit.

Because a configuration file may contain multiple `<persistence_service>` tags, one file can be used to configure multiple Persistence Service executions.

Table 46.2 Persistence Service Application Tags lists the tags you can specify for a persistence service. Notice that `<participant>` is required. For default values, please see the API Reference HTML documentation.

### Table 46.2 Persistence Service Application Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;persistence_service&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;administration&gt;</code></td>
<td>Enables and configures remote administration. See 46.5 Configuring Remote Administration on the next page.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| `<annotation>`                    | Provides a description for the persistence service configuration. Example: `<annotation>
  <documentation>
    Persists in the file system all topics published with PERSISTENT durability
  </documentation>
</annotation>` | 0 or 1 |

For more information about QoS libraries and profiles see Configuring QoS with XML (Chapter 19 on page 848).
Table 46.2 Persistence Service Application Tags

<table>
<thead>
<tr>
<th>Tags within &lt;persistence_service&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;purge_samples_after_acknowledgment&gt;</td>
<td>A DDS_Boolean that indicates whether or not a PRSTDataWriter will purge a DDS sample from its cache once it is acknowledged by all the matching/active DataReaders and all the Durable Subscriptions. Default: 0 See 46.9 Configuring Durable Subscriptions in Persistence Service on page 1200.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| <participant> | For each <participant> tag, Persistence Service creates two DomainParticipants on the same domain ID: one to subscribe to changes and one to publish changes. There may be more Participant pairs created when there are multiple versions of a type (see 46.13 Support for Extensible Types on page 1204). The QoS values used to configure both DomainParticipants are the same, except for:  
  - The participant_id in the 9.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) on page 670. If participant_id is not -1 (the default value, which means automatic selection), Persistence Service uses participant_id for the first DomainParticipant and participant_id+1 for the second DomainParticipant.  
  - The TCP server ports are configured with the properties dds.transport.tcp.server_bind_port and dds.transport.tcp.public_address. See 44.7 TCP/TLS Transport Properties on page 1158. | 1 or more (required) |
| <persistent_storage> | When this tag is present, the topic data will be persisted to disk. You can select between file storage and relational database storage. See 46.6 Configuring Persistent Storage on the next page. | 0 or 1 |
| <synchronization> | Enables synchronization in redundant persistence service instances. See 46.10 Synchronizing of Persistence Service Instances on page 1201. Default: Synchronization is not enabled | 0 or 1 |

46.5 Configuring Remote Administration

You can create a Connext DDS application that can remotely control Persistence Service. The <administration> tag is used to enable remote administration and configure its behavior.

By default, remote administration is turned off in Persistence Service.

When remote administration is enabled, Persistence Service will create a DomainParticipant, Publisher, Subscriber, DataWriter, and DataReader. These Entities are used to receive commands and send responses. You can configure these entities with QoS tags within the <administration> tag.

Table 46.3 Remote Administration Tags lists the tags allowed within <administration> tag. Notice that the <domain_id> tag is required.

For more details, please see Administering Persistence Service from a Remote Location (Chapter 48 on page 1211).

Note: The command-line options used to configure remote administration take precedence over the XML configuration (see Table 47.1 Persistence Service Command-Line Options).
### Table 46.3 Remote Administration Tags

<table>
<thead>
<tr>
<th>Tags within &lt;administration&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;datareader_qos&gt;</td>
<td>Configures the DataReader QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults with the following changes: reliability.kind = DDS_RELIABLE_RELIABILITY_QOS (this value cannot be changed) history.kind = DDS_KEEP_ALL_HISTORY_QOS resource_limits.max_samples = 32</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;datawriter_qos&gt;</td>
<td>Configures the DataWriter QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults with the following changes: history.kind = DDS_KEEP_ALL_HISTORY_QOS resource_limits.max_samples = 32</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;distributed_logger&gt;</td>
<td>Configures RTI Distributed Logger.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;domain_id&gt;</td>
<td>Specifies which domain ID Persistence Service will use to enable remote administration.</td>
<td>1 (required)</td>
</tr>
<tr>
<td>&lt;participant_qos&gt;</td>
<td>Configures the DomainParticipant QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;publisher_qos&gt;</td>
<td>Configures the Publisher QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;subscriber_qos&gt;</td>
<td>Configures the Subscriber QoS for remote administration. If the tag is not defined, Persistence Service will use the DDS defaults.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

### 46.6 Configuring Persistent Storage

The `<persistent_storage>` tag is used to persist DDS samples into permanent storage. If the `<persistent_storage>` tag is not specified, the service will operate in TRANSIENT mode and all the data will be kept in memory. Otherwise, the persistence service will operate in PERSISTENT mode and all the topic data will be stored into the filesystem or into a relational database that provides an ODBC driver.

Table 46.4 Persistent Storage tags lists the tags that you can specify in `<persistent_storage>`.

Relational Database Limitations: The ODBC storage does not support BLOBs. The maximum size for a serialized DDS sample is 65535 bytes in MySQL.
### Table 46.4 Persistent Storage tags

<table>
<thead>
<tr>
<th>Tags within &lt;persistent_storage&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPRECATED &lt;external_database&gt;</td>
<td>When this tag is present, the topic data will be persisted in a relational database. This tag is required if &lt;filesystem&gt; is not specified. See Table 46.5 External Database Tags.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;filesystem&gt;</td>
<td>When this tag is present, the topic data will be persisted into files. This tag is required if &lt;external_database&gt; is not specified. See Table 46.6 Filesystem tags.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;restore&gt;</td>
<td>This DDS_Boolean (see Table 46.1 Supported Tag Values) indicates if the topic data associated with a previous execution of the persistence service must be restored or not. If the topic data is not restored, it will be deleted from the persistent storage. Default: 1</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;type_object_max_serialized_length&gt;</td>
<td>Defines the length in bytes of the database column used to store the TypeObjects associated with PRSTDataWriters and PRSTDataReader. For additional information on TypeObjects, see the RTI Connext DDS Core Libraries Extensible Types Guide. Default: 10488576</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

### Table 46.5 External Database Tags

<table>
<thead>
<tr>
<th>Tags within &lt;external_database&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEPRECATED &lt;dsn&gt;</td>
<td>DSN used to connect to the database using ODBC. You should create this DSN through the ODBC settings on Windows systems, or in your .odbc.ini file on Linux systems. This tag is required.</td>
<td>1 (required)</td>
</tr>
<tr>
<td>DEPRECATED &lt;odbc_library&gt;</td>
<td>Specifies the ODBC driver to load. By default, Connext DDS will try to use the standard ODBC driver manager library (UnixOdbc on Linux systems, the Windows ODBC driver manager on Windows systems).</td>
<td>0 or 1</td>
</tr>
<tr>
<td>DEPRECATED &lt;password&gt;</td>
<td>Password to connect to the database. Default: no username is used</td>
<td>0 or 1</td>
</tr>
<tr>
<td>DEPRECATED &lt;username&gt;</td>
<td>Username to connect to the database. Default: no username is used</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
### Table 46.6 Filesystem tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;filesystem&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;directory&gt;</code></td>
<td>Specifies the directory of the files in which topic data will be persisted. There will be one file per PRSTDataWriter/PRSTDataReader pair. The directory must exist; otherwise the service will report an error upon start up. Default: current working directory</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;file_prefix&gt;</code></td>
<td>A name prefix associated with all the files created by Persistence Service. Default: PS</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| `<journal_mode>`          | Sets the journal mode of the persistent storage. This tag can take these values:  
  - DELETE: Deletes the rollback journal at the conclusion of each transaction.  
  - TRUNCATE: Commits transactions by truncating the rollback journal to zero-length instead of deleting it.  
  - PERSIST: Prevents the rollback journal from being deleted at the end of each transaction. Instead, the header of the journal is overwritten with zeros.  
  - MEMORY: Stores the rollback journal in volatile RAM. This saves disk I/O.  
  - WAL: Uses a write-ahead log instead of a rollback journal to implement transactions.  
  - OFF: Completely disables the rollback journal. If the application crashes in the middle of a transaction when the OFF journaling mode is set, the files containing the DDS samples will very likely be corrupted. Default: DELETE | 0 or 1 |
| `<synchronization>`      | Determines the level of synchronization with the physical disk. This tag can take three values:  
  - FULL: Every DDS sample is written into physical disk as Persistence Service receives it.  
  - NORMAL: DDS samples are written into disk at critical moments.  
  - OFF: No synchronization is enforced. Data will be written to physical disk when the OS flushes its buffers. Default: OFF | 0 or 1 |
| `<trace_file>`            | Specifies the name of the trace file for debugging purposes. The trace file contains information about all SQL statements executed by the persistence service. Default: No trace file is generated | 0 or 1 |
| `<vacuum>`                | Sets the auto-vacuum status of the storage. This tag can take these values:  
  - NONE: When data is deleted from the storage files, the files remain the same size.  
  - FULL: The storage files are compacted every transaction. Default: FULL | 0 or 1 |
46.7 Configuring Participants

An XML `<persistence_service>` tag will contain a set of `<participants>`. The persistence service will persist topics published in the domainIDs associated with these participants. For example:

```xml
<persistence_service name="Srv1">
  <participant name="Part1">
    <domain_id>71</domain_id>
    ...
  </participant>
  <participant name="Part2">
    <domain_id>72</domain_id>
    ...
  </participant>
</persistence_service>
```

Using the above example, the persistence service will create two pairs of `DomainParticipants` on DDS domains 71 and 72, respectively. In each pair, one `DomainParticipant` is used to receive data and the other to publish.

After the `DomainParticipants` are created, the persistence service will monitor the discovery traffic, looking for topics to persist.

Notice that in some cases there may be more than one pair of `DomainParticipants` per domain when there are multiple versions of a type for a given topic. (See 46.13 Support for Extensible Types on page 1204.)

The `<domain_id>` tag can be specified alternatively as an attribute of `<participant>`. For example:

```xml
<persistence_service name="Srv1">
  <participant name="Part1" domain_id="71">
    ...
  </participant>
</persistence_service>
```

**Table 46.7 Participant Tags** describes the participant tags. Notice that `<persistence_group>` is required.

**Table 46.7 Participant Tags**

<table>
<thead>
<tr>
<th>Tags within <code>&lt;participant&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;domain_id&gt;</code></td>
<td>Domain ID associated with the Participant. The domain ID can be specified as an attribute of the participant tag. Default: 0</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
### 46.8 Creating Persistence Groups

The topics that must be persisted in a specific domain ID are specified using `<persistence_group>` tags. A `<persistence_group>` tag defines a set of topics identified by a POSIX expression.

For example:

```xml
<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="H*">
    ...
  </persistence_group>
</participant>
```

In the above example, the persistence group 'PerGroup1' is associated with all the topics published in DDS domain 71 whose name starts with 'H'.

When a participant discovers a topic that matches a persistence group, it will create a PRSTDataReader and a PRSTDataWriter. The PRSTDataReader and PRSTDataWriter will be configured using the QoS policies associated with the persistence group. The DDS samples received by the PRSTDataReader will be persisted in the queue of the corresponding PRSTDataWriter.
A `<participant>` tag can contain multiple persistence groups; the set of topics that each one represents can intersect.

Table 46.8 Persistence Group Tags further describes the persistence group tags. For default values, please see the API Reference HTML documentation.

### Table 46.8 Persistence Group Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;persistence_group&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;allow_durable_subscriptions&gt;</code></td>
<td>A DDS_Boolean (see Table 46.1 Supported Tag Values) that enables support for durable subscriptions in the PRSTDataWriters created in a persistence group. When Durable Subscriptions are not required, setting this property to 0 will increase performance. Default: 1</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;content_filter&gt;</code></td>
<td>Content filter topic expression. A persistence group can subscribe to a specific set of data based on the value of this expression. A filter expression is similar to the WHERE clause in SQL. For more information on the syntax, please see the API Reference Documentation (from the Modules page, select RTI Connext DDS DDS API Reference, Queries and Filters Syntax). Default: no expression</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;datareader_qos&gt;</code></td>
<td>PRSTDataReader QoS¹. See 46.8.1 QoSs on page 1197. Default: DDS defaults</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;datawriter_qos&gt;</code></td>
<td>PRSTDataWriter QoS². See 46.8.1 QoSs on page 1197. Default: DDS defaults</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;deny_filter&gt;</code></td>
<td>Specifies a list of POSIX expressions separated by commas that describe the set of topics to be denied in the persistence group. This &quot;black&quot; list is applied to the topics that pass the filter specified with the <code>&lt;filter&gt;</code> tag. Default: *</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;filter&gt;</code></td>
<td>Specifies a list of POSIX expressions separated by commas that describe the set of topics associated with the persistence group. The filter can be specified as an attribute of <code>&lt;persistence_group&gt;</code> as well. Default: *</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;memory_management&gt;</code></td>
<td>This flag configures the memory allocation policy for DDS samples in PRSTDataReaders and PRSTDataWriters. See 46.8.5 Memory Management on page 1199.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

¹These fields cannot be set and are assigned automatically: `protocol.virtual_guid`, `protocol.rtps_object_id`, `durability.kind`.

²These fields cannot be set and are assigned automatically: `protocol.virtual_guid`, `protocol.rtps_object_id`, `durability.kind`.
### Table 46.8 Persistence Group Tags

<table>
<thead>
<tr>
<th>Tags within &lt;persistence_group&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;propagate_dispose&gt;</td>
<td><strong>A DDS.Boolean</strong> (see Table 46.1 Supported Tag Values) that controls whether or not the persistence service propagates dispose messages from DataWriters to DataReaders. When this tag is set to true, Persistence Service propagates dispose samples, but it doesn't propagate the key value for those dispose samples. Therefore, a DataReader receiving samples from the Persistence Service may not be able to access the key value for an instance if the first and only sample that it received is a dispose sample of that instance. Default: 1</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;propagate_source_timestamp&gt;</td>
<td><strong>A DDS.Boolean</strong> (see Table 46.1 Supported Tag Values). When this tag is 1, the DDS data samples sent by the PRSTDataWriters preserve the source timestamp that was associated with them when they were published by the original DataWriter. Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;propagate_unregister&gt;</td>
<td><strong>A DDS.Boolean</strong> (see Table 46.1 Supported Tag Values) that controls whether or not the persistence service propagates unregister messages from DataWriters to DataReaders. Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;publisher_qos&gt;</td>
<td>Publisher QoS. See 46.8.1 QoSs on page 1197. Default: DDS defaults</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;reader_checkpoint_frequency&gt;</td>
<td>This property controls how often (expressed as a number of DDS samples) the PRSTDataReader state is stored in the database. The PRSTDataReaders are the DataReaders created by the persistence service. A high frequency will provide better performance. However, if the persistence service is restarted, it may receive some duplicate DDS samples. The persistence service will send these duplicates DDS samples on the wire but they will be filtered by the DataReaders and they will not be propagated to the application. This property is only applicable when the persistence service operates in persistent mode (the &lt;persistent_storage&gt; tag is present). Default: 1</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;share_database_connection&gt;</td>
<td><strong>A DDS.Boolean</strong> (see Table 46.1 Supported Tag Values) that indicates if the persistence service will create an independent database connection per PRSTDataWriter in the group (0) or per Publisher (1) in the group. When &lt;single_publisher&gt; is 0 and &lt;share_database_connection&gt; is 1, there is a single database connection per group. All the PRSTDataWriters will share the same connection. When &lt;single_publisher&gt; is 1 or &lt;share_database_connection&gt; is 0, there is a database connection per PRSTDataWriter. This parameter is only applicable to configurations persisting the data into a relational database using the tag &lt;external_database&gt; in &lt;persistent_storage&gt;. See 46.8.4 Sharing a Database Connection on page 1199 Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;single_publisher&gt;</td>
<td><strong>A DDS.Boolean</strong> (see Table 46.1 Supported Tag Values) that indicates if the persistence service should create one Publisher per persistence group or one Publisher per PRSTDataWriter inside the persistence group. See 46.8.3 Sharing a Publisher/Subscriber on page 1198 Default: 1</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
### Table 46.8 Persistence Group Tags

<table>
<thead>
<tr>
<th>Tags within &lt;persistence_group&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;single_subscriber&gt;</td>
<td>A DDS_Boolean (see Table 46.1 Supported Tag Values) that indicates if the persistence service should create one Subscriber per persistence group or one Subscriber per PRSTDataReader in the persistence group. See 46.8.3 Sharing a Publisher/Subscriber on page 1198. Default: 1</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;subscriber_qos&gt;</td>
<td>Subscriber QoS. See 46.8.1 QoSs on the next page. Default: DDS defaults</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;topic_qos&gt;</td>
<td>Topic QoS. See 46.8.1 QoSs on the next page. Default: DDS defaults</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;use_durability_service&gt;</td>
<td>A DDS_Boolean (see Table 46.1 Supported Tag Values) that indicates if the HISTORY and RESOURCE_LIMITS QoS policy of the PRSTDataWriters and PRSTDataReaders should be configured based on the DURABILITY SERVICE value of the discovered DataWriters. See 46.8.2 DurabilityService QoS Policy on page 1198 Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;writer_ack_period&gt;</td>
<td>Controls how often (expressed in milliseconds) DDS samples are marked as ACK'd in the database by the PRSTDataWriter. Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;writer_checkpoint_period&gt;</td>
<td>Controls how often (expressed in milliseconds) transactions are committed for a PRSTDataWriter. A value of 0 indicates that transactions will be committed immediately. This is the recommended setting to avoid losing data in the case of an unexpected error in Persistence Service and/or the underlying hardware/software infrastructure. For applications that can tolerate some data losses, setting this tag to a value greater than 0 will increase performance. Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;writer_checkpoint_volume&gt;</td>
<td>Controls how often (expressed as a number of DDS samples) transactions are committed for a PRSTDataWriter. A value of 1 indicates that DDS samples will be persisted by the PRSTDataWriters immediately. This is the recommended setting to avoid losing data in the case of an unexpected error in persistence service and/or the underlying hardware/software infrastructure. For application that can tolerate some data losses, setting this tag to a value greater than 1 will increase performance. Default: 1</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;late_joiner_read_batch&gt;</td>
<td>Defines how many DDS samples will be pre-fetched by a PRSTDataWriter to satisfy requests from late-joiners. When a DataReader requests DDS samples from a PRSTDataWriter by sending a NACK message, the PRSTDataWriter may retrieve additional DDS samples from the database to minimize disk access. This parameter determines that amount of DDS samples that will be retrieved preemptively from the database by the PRSTDataWriter. Default: 20000</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
Table 46.8 Persistence Group Tags

<table>
<thead>
<tr>
<th>Tags within &lt;persistence_group&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;sample_logging&gt;</td>
<td>This tag can be used to enable and configure a DDS sample log for the PRSTDataWriters in a persistence group. A DDS sample log is a buffer of DDS samples on disk that, when used in combination with delegate reliability, allow decoupling the original DataWriters from slow DataReaders. For additional information on the DDS sample log, see 49.3 Scenario: Slow Consumer on page 1220. Default: DDS sample log is disabled.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;writer_in_memory_state&gt;</td>
<td>A DDS_Boolean (see Table 46.1 Supported Tag Values) that determines how much state will be kept in memory by the PRSTDataWriters in order to avoid accessing the persistent storage. The property is only applicable when the persistence service operates in persistent mode (the &lt;persistent_storage&gt; tag is present). If this property is 1, the PRSTDataWriters will keep a copy of all the instances in memory. They will also keep a fixed state overhead of 24 bytes per DDS sample. This mode provides the best performance. However, the restore operation will be slower and the maximum number of DDS samples that a PRSTDataWriter can manage will be limited by the available physical memory. If this property is 0, all the state will be kept in the underlying persistent storage. In this mode, the maximum number of DDS samples that a PRSTDataWriter can manage will not be limited by the available physical memory. Default: If the HistoryQosPolicy’s kind is KEEP_LAST or the ResourceLimitsQosPolicy’s max_samples != DDS_UNLIMITED_LENGTH, the default is 1. Otherwise, the default is 0.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;use_wait_set&gt;</td>
<td>A DDS_Boolean (see Table 46.1 Supported Tag Values) that indicates if Persistence Service will use Waitsets or Listeners to read data from the PRSTDataReaders of the group. By default, the usage of Waitsets is disabled. With this configuration, Persistence Service uses the on_data_available() listener callback to take the data from the PRSTDataReaders within the persistence group. The write operation in a PRSTDataWriter is called within the listener callback. When Waitsets are enabled, Persistence Service will use them to read the data: If &lt;single_subscriber&gt; is set to 1, there will be a single Waitset and a read thread shared across all the PRSTDataReaders in the group. If &lt;single_subscriber&gt; is set to 0, there will be a Waitset and a read thread per PRSTDataReader in the group. The write operation in a PRSTDataWriter is called by the read thread associated with the PRSTDataReader. Default: 0</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

46.8.1 QoSs

When a persistence service discovers a topic 'A' that matches a specific persistence group, it creates a reader (known as ‘PRSTDataReader’) and writer (‘PRSTDataWriter’) to persist that topic. The QoSs associated with these readers and writers, as well as the corresponding publishers and subscribers, can be configured inside the persistence group using QoS tags.

For example:

```xml
<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <publisher_qos base_name="QosLib1::PubQos1"/>
    <subscriber_qos base_name="QosLib1::SubQos1"/>
    <datawriter_qos base_name="QosLib1::WriterQos1"/>
```
For instance, the number of DDS samples saved by Persistence Service is configurable through the 7.5.12 HISTORY QosPolicy on page 420 of the PRSTDataWriters.

If a QoS tag is not specified, the persistence service will use the corresponding DDS default values (46.8.2 DurabilityService QoS Policy below describes an exception to this rule).

### 46.8.1.1 DataRepresentation QoS Policy

The PRSTDataReader's DataRepresentation QoS Policy may contain either XCDR_DATA_REPRESENTATION or XCDR2_DATA_REPRESENTATION, but not both. The PRSTDataReader and PRSTDataWriter of a given topic must have identical DataRepresentation QoS Policy values. See 7.5.3 DATA_REPRESENTATION QosPolicy on page 380.

### 46.8.2 DurabilityService QoS Policy

The 7.5.10 DURABILITY SERVICE QosPolicy on page 416 associated with a DataWriter is used to configure the HISTORY and the RESOURCE_LIMITS associated with the PRSTDataReaders and PRSTDataWriters.

By default, the HISTORY and RESOURCE_LIMITS of a PRSTDataReader and PRSTDataWriter with topic 'A' will be configured using the values specified in the XML file used to configure Persistence Service. To overwrite those values and use the values in the 7.5.10 DURABILITY SERVICE QosPolicy on page 416 of the first discovered DataWriter publishing 'A', you can use the tag <use_durability_service> in the persistence group definition:

```xml
<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <use_durability_service/>
    ...
  </persistence_group>
</participant>
```

### 46.8.3 Sharing a Publisher/Subscriber

By default, the PRSTDataWriters and PRSTDataReaders associated with a persistence group will share the same Publisher and Subscriber.

To associate a different Publisher and Subscriber with each PRSTDataWriter and PRSTDataReader, use the tags <single_publisher> and <single_subscriber>, as follows:

```xml
<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <single_publisher/>
    ...
    <single_subscriber/>
  </persistence_group>
</participant>
```
46.8.4 Sharing a Database Connection

By default, the persistence service will share a single ODBC database connection to persist the topic data received by each PRSTDataReader.

To associate an independent database connection to the PRSTDataReaders created by the persistence service, use the tag `<share_database_connection>`, as follows:

```
<participant name="Part1">
  <domain_id>71</domain_id>
  <persistence_group name="PerGroup1" filter="*">
    ...
    <share_database_connection>0</share_database_connection>
    ...
  </persistence_group>
</participant>
```

Sharing a database connection optimizes the resource usage. However, the concurrency of the system decreases because the access to the database connection must be protected.

46.8.5 Memory Management

The DDS samples received and stored by the PRSTDataReaders and PRSTDataWriters are in serialized form.

The serialized size of a DDS sample is the number of bytes required to send the DDS sample on the wire. The maximum serialized size of a DDS sample is the number of bytes that the largest DDS sample for a given type requires on the wire.

By default, the PRSTDataReaders and PRSTDataWriters created by the persistence service try to allocate multiple DDS samples to their maximum serialized size. This may cause memory allocation issues when the maximum serialized size is significantly large.

For PRSTDataReaders, the number of DDS samples in the `DataReader`'s queues can be controlled using the QoS values `resource_qos.resource_limits.max_samples` and `resource_qos.resource_limits.initial_samples`.

The PRSTDataWriters keep a cache of DDS samples so that they do not have to access the database every time. The minimum size of this cache is 32 DDS samples.

In addition, each PRSTDataWriter keeps an additional DDS sample called the DB sample, which is used to move information from the `DataWriter` cache to the database and vice versa.
The `<memory_management>` tag in a persistence group can be used to control the memory allocation policy for the DDS samples created by PRSTDDataReaders and PRSTDDataWriters in the persistence group.

Table 46.9 Memory Management Tags describes the memory management tags.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;memory_management&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;persistent_sample_buffer_max_size&gt;</code></td>
<td>This tag is used to control the memory associated with the DB sample in a PRSTDDataWriter. The persistence service will not be able to store a DDS sample into persistent storage if the serialized size is greater than this value. Therefore, this parameter must be used carefully. Default: LENGTH_UNLIMITED (DB sample is allocated to the maximum size).</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;pool_sample_buffer_max_size&gt;</code></td>
<td>This tag applies to both PRSTDDataReaders and PRSTDDataWriters. Its value determines the maximum size (in bytes) of the buffers that will be pre-allocated to store the DDS samples. If the space required for a new DDS sample is greater than this size, the persistence service will allocate the memory dynamically to the exact size required by the DDS sample. This parameter is used to control the memory allocated for the DDS samples in the PRSTDDataReaders queues and the PRSTDDataWriters caches. The size of the DB sample in the PRSTDDataWriters is controlled by the value of the tag <code>&lt;persistent_sample_buffer_max_size&gt;</code>. Default: 4096</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

### 46.9 Configuring Durable Subscriptions in Persistence Service

This section assumes you are familiar with the concept of 7.3.13 Required Subscriptions on page 323.

A Durable Subscription is a Required Subscription where DDS samples are stored and forwarded by Persistence Service.

There are two ways to create a Durable Subscriptions:

1. Programmatically using a `DomainParticipant` API:

   A subscribing application can register a Durable Subscription by providing the topic name and the durable subscription information, consisting of the Durable Subscription `name` (role _name in `DDS_EndpointGroup_t`) and the `quorum_count` (quorum_count in `DDS_EndpointGroup_t`). To register or delete a Durable Subscription, use the `DomainParticipant`’s `register_durable_subscription()` and `delete_durable_subscription()` operations, respectively (see Table 9.3 DomainParticipant Operations). The Durable Subscription information is propagated via a built-in topic to Persistence Service.

2. Preconfigure `Persistence Service` with a set of Durable Subscriptions:
Persistence Service can be (pre-)configured with a list of Durable Subscriptions using the <durable_subscriptions> XML tag under <participant>.

```
<participant name="Participant">
... 
<durable_subscriptions>
   <element>
      <name>Logger</name>
      <topic_name>Track</topic_name>
      <quorum_count>2</quorum_count>
   </element>
   <element>
      <name>Processor</name>
      <topic_name>Track</topic_name>
      <quorum_count>1</quorum_count>
   </element>
('</durable_subscriptions>
</participant>
```

After registering or configuring the persistence service with specific Durable Subscriptions, the persistence service will keep DDS samples until they are acknowledged by all the required Durable Subscriptions. In the above example, the DDS samples must be acknowledged by two DataReaders that belong to the “Logger” Durable Subscription and one DataReader belonging to the “Processor” Durable Subscription.

### 46.9.1 DDS Sample Memory Management With Durable Subscriptions

The maximum number of DDS samples that will be kept in a PRSTDataWriter queue is determined by the value of <resource_limits><max_samples> in the <writer_qos> used to configure the PRSTDataWriter.

By default, a PRSTDataWriter configured with KEEP_ALL <history><kind> will keep the DDS samples in its cache until they are acknowledged by all the Durable Subscriptions associated with the PRSTDataWriter. After the DDS samples are acknowledged by the Durable Subscriptions, they will be marked as reclaimable but they will not be purged from the PRSTDataWriter’s queue until the DataWriter needs these resources for new DDS samples. This may lead to inefficient resource utilization, especially when <max_samples> is high or UNLIMITED.

The PRSTDataWriter behavior can be changed to purge DDS samples after they have been acknowledged by all the active/matching DataReaders and all the Durable Subscriptions configured for the <persistence_service>. To do so, set the tag <purge_samples_after_acknowledgment> under <persistence_service> to TRUE. Notice that this setting is global to the service and applies to all the PRSTDataWriters created by each <persistence_group>.

### 46.10 Synchronizing of Persistence Service Instances

By default, different Persistence Service instances do not synchronize with each other. For example, in a scenario with two Persistence Service instances, the first persistence service could receive a DDS sample
Enabling RTI Distributed Logger in Persistence Service

`S1` from the original *DataWriter* that is not received by the second persistence service. If the disk where the first persistence service stores its DDS samples fails, `S1` will be lost.

To enable synchronization between *Persistence Service* instances, use the tag `<synchronization>` under `<persistence_service>`. When it comes to synchronization, there are two different kinds of information that can be synchronized independently:

- Information about Durable Subscriptions and their states (see 46.9 Configuring Durable Subscriptions in Persistence Service on page 1200)
- DDS data samples

Table 46.10 Synchronization Tags

<table>
<thead>
<tr>
<th>Tags within &lt;synchronization&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;synchronize_data&gt;</td>
<td>Enables synchronization of DDS data samples in redundant <em>Persistence Service</em> instances. When set to 1, DDS samples lost on the way to one service instance can be repaired by another without impacting the original publisher of that message. To synchronize the instances, the tag <code>&lt;synchronize_data&gt;</code> must be set to 1 in every instance involved in the synchronization. Note: This DDS sample synchronization mechanism is not equivalent to database replication. The extent to which database instances have identical contents depends on the destination ordering and other QoS settings for the <em>Persistence Service</em> instances. Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;synchronize_durable_subscription&gt;</td>
<td>Enables synchronization of Durable Subscriptions in redundant <em>Persistence Service</em> instances. When set to 1, the different <em>Persistence Service</em> instances will synchronize their Durable Subscription information. This information includes the set of Durable Subscriptions as well as information about the Durable Subscription’s state, such as the DDS samples that have already been received by the Durable Subscriptions. Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;durable_subscription_synchronization_period&gt;</td>
<td>The period (in milliseconds) at which the information about Durable Subscriptions is synchronized. Default: 5000 milliseconds</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

46.11 Enabling RTI Distributed Logger in Persistence Service

*Persistence Service* provides integrated support for *RTI Distributed Logger* (see *Part 10: RTI Distributed Logger* on page 1239).

*Distributed Logger* is included in *Connext DDS* but it is not supported on all platforms; see the *RTI Connext DDS Core Libraries Platform Notes* to see which platforms support *Distributed Logger*. 

1202
When you enable Distributed Logger, Persistence Service will publish its log messages to Connext DDS. Then you can use RTI Monitor\(^1\) to visualize the log message data. Since the data is provided in a Connext DDS topic, you can also use rtiddsspy or even write your own visualization tool.

To enable Distributed Logger, modify the Persistence Service XML configuration file. In the <administration> section, add the <distributed_logger> tag as shown in the example below.

```xml
<persistence_service name="default">
    ...
    <administration>
        ...
        <distributed_logger>
            <enabled>true</enabled>
        </distributed_logger>
        ...
    </administration>
    ...
</persistence_service>
```

There are more configuration tags that you can use to control Distributed Logger’s behavior. For example, you can specify a filter so that only certain types of log messages are published. For details, see Enabling Distributed Logger in RTI Services (Chapter 53 on page 1250)

### 46.12 Enabling RTI Monitoring Library in Persistence Service

Persistence Service provides integrated support for RTI Monitoring Library (see Part 9: RTI Monitoring Library on page 1223).

To enable monitoring in Persistence Service, you must specify the property rti.monitor.library for the participants that you want to monitor. For example:

\(^1\)RTI Monitor is a separate GUI application that can run on the same host as your application or on a different host.
Since *Persistence Service* is statically linked with *RTI Monitoring Library*, you do *not* need to have it in your library search path.

For details on how to configure the monitoring process, see Configuring Monitoring Library (Chapter 51 on page 1234).

**46.13 Support for Extensible Types**

*Persistence Service* includes partial support for the OMG 'Extensible and Dynamic Topic Types for DDS' specification, version 1.3. This section assumes that you are familiar with Extensible Types and you have read the RTI Connext DDS Core Libraries Extensible Types Guide.

Persistence groups can publish and subscribe to topics associated with final, appendable, and mutable types.

### 46.13.1 TypeConsistencyEnforcementQosPolicy Integration

The service will automatically create different pairs (PRSTDataReader, PRSTDataWriter) for each version of a type discovered for a topic in a persistence group. In *Connext DDS* 5.0, it is not possible to associate more than one type with a topic within a single *DomainParticipant*, therefore each version of a type requires its own *DomainParticipant*.

The 8.6.6 TYPE_CONSISTENCY_ENFORCEMENT QosPolicy on page 594 **kind** for each PRSTDataReader is set to DISALLOW_TYPE_COERCION. This value cannot be overwritten by the user.

For example:

```c
struct A {
    int32 x;
};
struct B {
```
Let’s assume that Persistence Service is configured as follows and we have two DataWriters on Topic “T” publishing type “A” and type “B” and sending TypeObject information.

When Persistence Service discovers the first DataWriter with type “A”, it will create a DataReader (PRSTDataReader) to read DDS samples from that DataWriter, and a DataWriter (PRSTDataWriter) to publish and store the received DDS samples so they can be available to late-joiners.

When Persistence Service discovers the second DataWriter with type “B”, it will see that type “B” is not equal to type “A”; then it will create a new pair (PRSTDataReader, PRSTDataWriter) to receive and store DDS samples from the second DataWriter.

Since the PRSTDataReaders are created with the TypeConsistencyEnforcementQosPolicy’s kind set to DISALLOW_TYPE_COERCION, the PRSTDataReader with type “A” will not match the DataWriter with type “B”. Likewise, the PRSTDataReader with type “B” will not match the DataWriter with type “A”.

46.13.1.1 Type Version Discrimination

Persistence Service uses the rules described in the RTI Connext DDS Core Libraries Extensible Types Guide to decide whether or not to create a new pair (PRSTDataReader, PRSTDataWriter) when it discovers a DataWriter for a topic “T”.

For DataWriters created with previous Connext DDS releases, Persistence Service will select the first pair (PRSTDataReader, PRSTDataWriter) with a registered type name equal to the discovered registered type name since DataWriters created with previous Connext DDS releases (before 5.0) do not send TypeObject information.

46.13.2 DataRepresentationQosPolicy Integration

There are some restrictions on how the 7.5.3 DATA_REPRESENTATION QosPolicy on page 380 is configured for the PRSTDataReader and PRSTDataWriter in a persistence group:

- A PRSTDataReader cannot be configured to request two or more data representations. For example, it is not possible to request XCDR and XCDR2. Subscribing to a Topic in which data is published in XCDR and XCDR2 format requires creating two different persistence groups.
46.14 TCP Transport Support in Persistence Service

The data representation requested by a PRSTDataReader has to be equal to the data representation offered by the corresponding PRSTDataWriter.

You can configure Persistence Service's Participants to use the TCP Transport. To do so, enable the TCP Transport under the proper XML Persistence Service's `<participant_qos>` tag.

Make sure the string prefix passed in the property `dds.transport.load_plugins` is "`dds.transport.tcp`". For more information about how to enable the TCP Transport, please see 44.7 TCP/TLS Transport Properties on page 1158.

Note that the Persistence Service's `participant_qos` will be used at least by two Participants: one for sending data and another for receiving data. Consequently, at least two TCP Transport plugins will be instantiated when enabling the TCP Transport. In order to avoid port collisions, Persistence Service will automatically assign consecutive ports. For a base, it will use the values set for `dds.transport.tcp.server_bind_port` (only when it is non-zero) and `dds.transport.tcp.public_address` (only if it is set). Consequently, the Participants creating a TCP Transport running as a server will open a minimum of two TCP ports.
Chapter 47 Running RTI Persistence Service

This chapter describes how to start and stop Persistence Service.

You can run Persistence Service on any node in the network. It does not have to be run on the same node as the publishing or subscribing applications for which it is saving/delivering data. If you run it on a separate node, make sure that the other applications can find it during the discovery process—that is, it must be in one of the NDDS_DISCOVERY_PEERS lists.

47.1 Starting Persistence Service

The script to run Persistence Service’s executable is located in <NDDSHOME>/bin.

To run this service executable on a target system (not your host development platform), you must first select the target architecture. To do so, either:

Set the environment variable CONNEXTDDS_ARCH to the name of the target architecture. (Do this for each command shell you will be using.)

Or set the variable connextdds_architecture in the file rticommon_config.[sh/bat] to the name of the target architecture. If the CONNEXTDDS_ARCH environment variable is set, the architecture in this file will be ignored.

Run rtipersistenceservice -help to see descriptions of the command-line options, which are also described in more detail in Table 47.1 Persistence Service Command-Line Options.

---

*aThis file is resource/scripts/rticommon_config.sh on Linux or macOS systems, resource/scripts/rticommon_config.bat on Windows systems.
## Table 47.1 Persistence Service Command-Line Options

<table>
<thead>
<tr>
<th>Command-line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-appName &lt;string&gt;</td>
<td>Assigns a name to the execution of Persistence Service. Remote commands will refer to the persistence service using this name. In addition, the name of the DomainParticipants created by Persistence Service will be based on this name as follows: RTI Persistence Service: &lt;appName&gt;: &lt;participantName&gt;: (&lt;pub</td>
</tr>
<tr>
<td>-cfgFile &lt;string&gt;</td>
<td>Specifies an XML configuration file for the Persistence Service. The parameter is optional since the Persistence Service configuration can be loaded from other locations. See 46.1 How to Load the Persistence Service XML Configuration on page 1182 for further details.</td>
</tr>
<tr>
<td>-cfgName &lt;string&gt;</td>
<td>Required. Selects a Persistence Service configuration. The same configuration files can be used to configure multiple persistence services. Each Persistence Service instance will load its configuration from a different &lt;persistence_service&gt; tag based on the name specified with this option. If not specified, Persistence Service will print the list of available configurations and then exit.</td>
</tr>
<tr>
<td>-identifyExecution</td>
<td>Appends the host name and process ID to the service name provided with the -appName option. This helps ensure unique names for remote administration.</td>
</tr>
<tr>
<td>-enableDatabaseLocking</td>
<td>Prevents multiple instances of Persistence Service from accessing the same database. This feature only has effect when &lt;persistent_storage&gt; is used. Default: Database locking is disabled. By default, multiple instances of Persistence Service can access the same database.</td>
</tr>
<tr>
<td>-domainId &lt;ID&gt;</td>
<td>Sets the domain ID for the DomainParticipants created by Persistence Service. If not specified, the value in the &lt;participant&gt; XML tag (see Table 46.7 Participant Tags) is used.</td>
</tr>
<tr>
<td>-remoteAdministrationDomainId &lt;ID&gt;</td>
<td>Enables remote administration and sets the domain ID for remote communication. When remote administration is enabled, Persistence Service will create a DomainParticipant, Publisher, Subscriber, DataWriter, and DataReader in the designated DDS domain. This option overwrites the value of the tag &lt;domain_id&gt; within &lt;administration&gt;. Default: Use the value &lt;domain_id&gt; under &lt;administration&gt;.</td>
</tr>
<tr>
<td>-help</td>
<td>Prints the Persistence Service version and list of command-line options.</td>
</tr>
<tr>
<td>-licenseFile &lt;file&gt;</td>
<td>Specifies the license file (path and filename). Only applicable to licensed-managed (LM) versions of Persistence Service. If not specified, Persistence Service looks for the license as described in the RTI Connext DDS Installation Guide.</td>
</tr>
<tr>
<td>-restore &lt;0</td>
<td>1&gt;</td>
</tr>
</tbody>
</table>
## Table 47.1 Persistence Service Command-Line Options

<table>
<thead>
<tr>
<th>Command-line Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-noAutoStart</code></td>
<td>Indicates that Persistence Service will not be started when the process is executed. Use this option if you plan to start Persistence Service remotely, as described in Administering Persistence Service from a Remote Location (Chapter 48 on page 1211).</td>
</tr>
<tr>
<td><code>-infoDir &lt;dir&gt;</code></td>
<td>The info directory of the running Persistence Service. Using this command line option, Persistence Service can be configured to create a file used to monitor the status of the last shutdown. At startup, the Persistence Service instance will create a file called <code>ps.pid</code> into the directory specified by <code>-infoDir</code>. If Persistence Service is shutdown gracefully, the file will be deleted before the process exists. You can detect the shutdown state of Persistence Service by checking for the presence of the <code>ps.pid</code> file. If the file is present and Persistence Service is no longer running, the previous shutdown was not graceful. If Persistence Service is started and a <code>ps.pid</code> file exists, Persistence Service will immediately shutdown. In this case, you must remove the file before Persistence Service can be restarted again. Default: The file <code>ps.pid</code> will not be generated.</td>
</tr>
<tr>
<td><code>-maxObjectsPerThread &lt;int&gt;</code></td>
<td>Parameter used to configure the maximum objects per thread in the DomainParticipantFactory created by Persistence Service. Default: Connext DDS default</td>
</tr>
<tr>
<td><code>-heapSnapshotPeriod</code></td>
<td>Enables heap monitoring. Persistence Service will generate a heap snapshot every ( n ) seconds. Default: Heap monitoring is disabled. Valid range: ([1, 86400])</td>
</tr>
<tr>
<td><code>-serviceThreadStackSize &lt;int&gt;</code></td>
<td>Service thread stack size. Default: Connext DDS default</td>
</tr>
<tr>
<td><code>-verbosity</code></td>
<td>Persistence Service verbosity: 0 - No verbosity 1 - Exceptions (Core Libraries and Persistence Service) 2 - Warning (Persistence Service) 3 - Information (Persistence Service) 4 - Warning (Core Libraries and Persistence Service) 5 - Tracing (Persistence Service) 6 - Tracing (Core Libraries and Persistence Service) Each verbosity level, ( n ), includes all the verbosity levels smaller than ( n ). Default: 1</td>
</tr>
<tr>
<td><code>-version</code></td>
<td>Prints the Persistence Service version.</td>
</tr>
</tbody>
</table>
47.2 Stopping Persistence Service

To stop Persistence Service: Press Ctrl-C.

Persistence Service will close all files and perform a clean shutdown. It can also be stopped and shutdown remotely (see Administering Persistence Service from a Remote Location (Chapter 48 on page 1211)).
Chapter 48 Administering Persistence Service from a Remote Location

Persistence Service can be controlled remotely by sending commands through a special Topic. Any Connext DDS application can be implemented to send these commands and receive the corresponding responses. A shell application that sends/receives these commands is provided with Persistence Service.

The script for the shell application is `SNDDS/Home/bin/rtipssh`.

Entering `rtipssh -help` will show you the command-line options:

```
RTI Persistence Service Shell v6.1.1
Usage: rtipssh [options]...
Options:
  -domainId  <integer>  Domain ID for the remote configuration
  -timeout   <seconds>  Max time to wait a remote response
  -cmdFile   <file>     Run commands in this file
  -help      Displays this information
```

### 48.1 Enabling Remote Administration

By default, remote administration is disabled in Persistence Service.

To enable remote administration you can use the `<administration>` tag (see 46.5 Configuring Remote Administration on page 1188) or the `-remoteAdministrationDomainId` command-line parameter (see Table 47.1 Persistence Service Command-Line Options), which enables remote administration and sets the domain ID for remote communication.

When remote administration is enabled, Persistence Service will create a DomainParticipant, Publisher, Subscriber, DataWriter, and DataReader in the designated DDS domain. (The QoS values for these entities are described in 46.5 Configuring Remote Administration on page 1188.)
48.2 Remote Commands

This section describes the remote commands using the shell interface; 48.3 Accessing Persistence Service from a Connext DDS Application on the next page explains how to use remote administration from a Connext DDS application.

Remote commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>start &lt;target_persistence_service&gt;</code></td>
<td>Starts the persistence service instance. DDS samples will not be persisted until the persistence service is started.</td>
</tr>
<tr>
<td><code>stop &lt;target_persistence_service&gt;</code></td>
<td>Stops the persistence service instance. An instance that has been stopped can be started again using the command start.</td>
</tr>
<tr>
<td><code>shutdown &lt;target_persistence_service&gt;</code></td>
<td>Stops the persistence service instance and finalizes the process.</td>
</tr>
</tbody>
</table>

Parameters:

*<target_persistence_service>* can be:

- The application name of a persistence service, such as “MyPersistenceService1”, as specified at start-up with the command-line option `-appName`
- A wildcard expression\(^1\) for a persistence service name, such as “MyPersistenceService*”

### 48.2.1 start

**start <target_persistence_service>**

The `start` command starts the persistence service instance. DDS samples will not be persisted until the persistence service is started.

By default, the persistence service is started automatically when the process is executed. To start the service remotely use the command line option `-noAutoStart` (see Table 47.1 Persistence Service Command-Line Options).

### 48.2.2 stop

**stop <target_persistence_service>**

The `stop` command stops the persistence service instance.

### 48.2.3 shutdown

**shutdown <target_persistence-service>**

The command `shutdown` stops the persistence service instance and finalizes the process.

---

\(^1\)As defined by the POSIX fnmatch API (1003.2-1992 section B.6)
48.2.4 status

The `status` command gets the status of a running persistence service instance. Possible values are STARTED and STOPPED.

### 48.3 Accessing Persistence Service from a Connext DDS Application

You can send commands to control a Persistence Service instance from your own Connext DDS application. You will need to create a `DataWriter` for a specific topic and type. Then, you can send a DDS sample that contains a command and its parameters. Optionally, you can create a `DataReader` for a specific topic to receive the results of the execution of your commands.

The topics are:

- `rti/persistence_service/administration/command_request`
- `rti/persistence_service/administration/command_response`

The types are:

- `RTI::PersistenceService::Administration::CommandRequest`
- `RTI::PersistenceService::Administration::CommandResponse`

You can find the IDL definitions for these types in:

`<NDDSHOME>/resource/idl/PersistenceServiceAdministration.idl`

The QoS configuration of your `DataWriter` and `DataReader` must be compatible with the one used by the persistence service (see how this QoS is configured in 46.5 Configuring Remote Administration on page 1188).

The following example in C shows how to send a command to shutdown a persistence service instance:

```c
/***********************************************************/
/** Create the Entities needed to send command request ****/
/***********************************************************/
participant = DDS_DomainParticipantFactory_create_participant(
    DDS_DomainParticipantFactory, domainId,
    &DDS_PARTICIPANT_QOS_DEFAULT, NULL,
    DDS_STATUS_MASK_NONE);
if (participant == NULL)
    { /* Error */ }
if (publisher == NULL)
    { /* Error */ }

subscriber = DDS_DomainParticipant_create_subscriber(
    participant, &DDS_SUBSCRIBER_QOS_DEFAULT,
    NULL, DDS_STATUS_MASK_NONE);
```
Accessing Persistence Service from a Connext DDS Application

```c
publisher = DDS_DomainParticipant_create_publisher(
    participant, &DDS_PUBLISHER_QOS_DEFAULT,
    NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL)
    { /* Error */ }

typeName = RTI_PersistenceService_Administration_CommandRequestTypeSupport_get_type_name();
rcode = RTI_PersistenceService_Administration_CommandRequestTypeSupport_register_type(
    participant, typeName);
if (rcode != DDS_RETCODE_OK)
    { /* Error */ }

topicCmd = DDS_DomainParticipant_create_topic(
    participant,
    "rti/persistence_service/administration/command_request",
    typeName, &DDS_TOPIC_QOS_DEFAULT,
    NULL, DDS_STATUS_MASK_NONE);
if (topicCmd == NULL)
    { /* Error */ }

typeName = RTI_PersistenceService_Administration_CommandResponseTypeSupport_get_type_name();
rcode = RTI_PersistenceService_Administration_CommandResponseTypeSupport_register_type(
    participant, typeName);
if (rcode != DDS_RETCODE_OK)
    { /* Error */ }

topicResponse = DDS_DomainParticipant_create_topic(
    participant,
    "rti/persistence_service/administration/command_response",
    typeName, &DDS_TOPIC_QOS_DEFAULT, NULL,
    DDS_STATUS_MASK_NONE);
if (topicResponse == NULL)
    { /* Error */ }

writerQos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
writerQos.history.kind = DDS_KEEP_ALL_HISTORY_QOS;
writer = DDS_Publisher_create_datawriter(
    publisher, topicCmd, &writerQos,
    NULL /* listener */,
    DDS_STATUS_MASK_NONE);
if (writer == NULL)
    { /* Error */ }

readerQos.reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
readerQos.history.kind = DDS_KEEP_ALL_HISTORY_QOS;
reader = DDS_Subscriber_create_datareader(
    subscriber,
    DDS_Topic_as_topicdescription(topicResponse),
    &readerQos, NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL)
    { /* Error */ }

/*******************************************************************/
```
/*** Wait for discovery **********************************************/
/*** Wait until we discover one reader and one writer matching  ***/
** with the command request DataWriter and the command response  ***/
** DataReader */
while (count < maxPollPeriods) {
    retcode = DDS_DataWriter_get_publication_matched_status(
        writer, &pubMatchStatus);
    if (retcode != DDS_RETCODE_OK) {
        /* Error */
    }
    retcode = DDS_DataReader_get_subscription_matched_status(
        reader, &subMatchStatus);
    if (retcode != DDS_RETCODE_OK) { /* Error */ }
    if (pubMatchStatus.total_count == 1 &&
        subMatchStatus.total_count == 1)
    { break; }
    count++;
    NDDS_Utility_sleep(&pollPeriod);
} if (count == maxPollPeriods) {
    /* Error */
}

/*****************************/
/**** Send the command request ********************************/}
/*****************************/
request =
    RTI_PersistenceService_Administration_CommandRequestTypeSupport_create_data();
if (request == NULL) {
    /* Error */
}
/* request->id provides an unique way to identify a request so that */
/* it can be correlated with a response. Although one of the fields is */
/* called host it does not necessarily has to contain the IP address of */
/* the host. Same applies to app */
request->id.host = 0;
request->id.app = 0;
request->id.invocation = 0;
strcpy(request->target_ps, "MyPersistenceService");
request->command._d = RTI_PERSISTENCE_SERVICE_COMMAND_SHUTDOWN;
retcode = RTI_PersistenceService_Administration_CommandRequestDataWriter_write(
    (RTI_PersistenceService_Administration_CommandRequestDataWriter *) writer,
    request, &instance_handle);
if (retcode != DDS_RETCODE_OK) {
    /* Error */
}

/*****************************/
/**** Wait for response *******************************************/
/*****************************/
response =
    RTI_PersistenceService_Administration_CommandResponseTypeSupport_create_data();
if (response == NULL) {
    /* Error */
    count = 0;
    while (count < maxPollPeriods) {

```c
retcode =
 RTI_PersistenceService_Administration_CommandResponseDataReader_take_next_sample(
 (RTI_PersistenceService_Administration_CommandResponseDataReader*) reader,
 response, &sampleInfo);
if (retcode == DDS_RETCODE_OK) {
  break;
} else if (retcode != DDS_RETCODE_NO_DATA) {
  /* Error */
}
NDDS_Utility_sleep(&pollPeriod);
count++;
}
if (count == maxPollPeriods) {
  printf("No response received\n");
} else {
  printf("Response received: %s\n", response->message);
}
```
Chapter 49 Advanced Persistence Service Scenarios

This section covers several advanced scenarios for using Persistence Service.

49.1 Scenario: Load-balanced Persistence Services

Each running instance of the Persistence Service executes as a single process in a single computer. In high-throughput scenarios the Persistence Service may become a bottleneck. The main reasons are:

- If the Persistence Service is configured to persist its DDS samples to durable storage (a disk or a database) this will further limit the throughput of DDS samples that can be persisted to what the database and/or disk can handle. Depending on computer hardware, the disk or database this limit may be in the order of tens of thousands of DDS samples per second which is far less than what could be communicated system-wide.

- Depending on the CPU there will be limits on the throughput of DDS samples that can be received by a single process.

- The computer running the Persistence Service is typically connected to the network via a single network interface so the data that can be persisted will be limited to the throughput that flows though a single interface which is typically far less that the aggregated throughput that can flow on the complete network.

To overcome these limits multiple instances of the RTI Persistence Service can be run in parallel. These instances may run in multiple machines and be configured in a “load balancing” fashion such that each Persistence Service process is only responsible for persisting a subset of the data published on the DDS domain.

Multiple strategies for partitioning the data stored by each Persistence Service instance are possible:
49.1 Scenario: Load-balanced Persistence Services

- **Balance Persistence Services by Topic name.** This strategy configures each persistence service to persist different Topic names. This is accomplished by associating a filter expression with the declaration of the persistent groups used to configure each Persistence Service (see 46.8 Creating Persistence Groups on page 1193). The filter expression is applied to the Topic names, so for example one Persistence Service could be configured with the filter “[A-Z]*” filter in the name of the Topics that it will persist and the second with the filter “[a-z]*”. With this configuration the first Persistence Service will persist data produced by DataWriters that specify durability TRANSIENT or PERSISTENT and have a Topic name that starts with a capital letter and the second Persistence Service will do the same for Topics that start with a lower-case letter.

- **Balance Persistence Services by data content.** In some scenarios the data published on a single Topic is too much for a single Persistence Service to handle. In this case the Persistence Services can also be configured with filter expressions based on the content of the data. This is accomplished by associating a content filter with the declaration of the persistent groups used to configure each Persistence Service (see 46.8 Creating Persistence Groups on page 1193).

When multiple instances of Persistence Service are used to store data on the same Topic, it becomes possible for DDS samples from the same original DataWriter to be stored in separate instances of Persistence Service. In this situation, Connext DDS DataReaders automatically merge the data from the multiple Persistence Services such that the relative order of the DDS samples from the original DataWriter is preserved. This Connext DDS capability is called Collaborative Datawriters because multiple DataWriters, in this case the ones for different Persistence Services, collaborate to reconstruct the original stream. (See Collaborative DataWriters (Chapter 12 on page 727)).

**Figure 49.1: Load-Balanced Persistence Services Scenario**

![Diagram showing load-balanced persistence services](image-url)
49.2 Scenario: Delegated Reliability

The DDS-RTPS reliability protocol requires the DataWriter to periodically send HeartBeat messages to the DataReaders, process their ACKs and NACK messages, keep track of the DataReader state, and send the necessary repairs. The additional load caused by the reliability protocol increases with the number of reliable DataReaders matched with the DataWriter. Even if the data is sent via multicast the number of ACKs and NACKs will increase with the number of DataReaders.

In situations where there many DataReaders are subscribing to the same Topic, the reliability and repair traffic may become too much for the DataWriter to handle and negatively impact its performance. To address this situation, Connext DDS provides the ability to configure the DataWriter so that it delegates the reliability task to a separate service. This approach is known as delegated reliability.

To take advantage of delegated reliability, both the original DataWriter and DataReader must be configured to enable an external service to ensure the reliability on their behalf. This is done by setting both the dds.data_writer.reliability.delegate_reliability property on the DataWriter and the dds.data_reader.reliability.delegate_reliability property on the DataReader to 1.

With this configuration, the DataWriter creates a reliable channel to Persistence Service, yet sends data using ‘best-effort’ reliability to the DataReaders directly. If a DDS sample is dropped, Persistence Service will repair the DDS sample. Persistence Service is configured with push_on_write (in the 7.5.5 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) on page 389) set to false. This way, DDS samples will only be sent from Persistence Service to the DataReaders when they are explicitly NACKed by the DataReader.

Figure 49.2: Delegated Reliability Scenario
49.3 Scenario: Slow Consumer

Unless special measures are taken, the presence of slow consumers can impact the overall behavior of the system. If a DataReader is not keeping up with the DDS samples being sent by the DataWriter, it will apply back-pressure to the DataWriter to slow the rate at which the DataWriter can write DDS samples. With delegated reliability (see 49.2 Scenario: Delegated Reliability on the previous page), the original DataWriter can offload the processing of the ACK/NACK messages generated by the DataReaders to a PRSTDataWriter. However, the original DataWriter still has a reliable channel with the PRSTDataReader that can slow it down.

By default, Persistence Service uses the Connext DDS receive thread to read DDS samples from the PRStDataReaders, write the DDS samples to the PRSTDataWriters history, and send ACKs to the original DataWriter. With this configuration, a PRSTDataReader does not ACK DDS samples to the original DataWriter until they are written into the corresponding PRSTDataWriter’s history. Since multiple DataReaders may be accessing the PRSTDataWriter history at the same time that the persistence service is trying to write new DDS samples, the PRSTDataWriter history becomes a contention point that can indirectly slow down the original DataWriter (see Figure 49.3: Slow-Consumer Scenario with Delegated Reliability below).

Figure 49.3: Slow-Consumer Scenario with Delegated Reliability

To remove this contention point and decouple the slow consumer from the original DataWriter, Persistence Service supports a mode where DDS samples can be buffered prior to being added to the PRSTDataWriter’s queue (see Figure 49.4: Slow Consumer Scenario with Delegated Reliability and DDS Sample Log on the next page).
49.3 Scenario: Slow Consumer

If the PRSTDataWriter slows down due to the presence of slow consumers, the buffer will hold DDS samples such that the original DataWriter and the rest of the system are not impacted. This buffer is called the Persistence Service sample log. The persistence service creates a separate DDS sample log per PRSTDataWriter in the group. In addition to the DDS sample log, the persistence service creates a thread (write thread) whose main function is to read DDS samples from the log and write them to the associated PRSTDataWriter. There is one thread per PRSTDataWriter.

Persistence Service currently does not allow multiple DDS sample logs to share the same write thread.

Persistence Service can be configured to enable DDS sample logging per persistence group using the <sample_logging> XML tag to specify the log’s configuration parameters—see Table 49.1 Sample Logging Tags.
### Table 49.1 Sample Logging Tags

<table>
<thead>
<tr>
<th>Tags within &lt;sample_logging&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enable&gt;</td>
<td>A DDS_Boolean (see Table 46.1 Supported Tag Values) that indicates whether or not DDS sample logging is enabled in the container persistence group. Default: 0</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;log_file_size&gt;</td>
<td>Specifies the maximum size of a DDS sample log file in Mbytes. When a log file becomes full, Persistence Service creates a new log file. Default: 60 MB</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;log_flush_period&gt;</td>
<td>The period (in milliseconds) at which Persistence Service removes DDS sample log files whose full content have been written into the PRSTDataWriter by the DDS sample log write thread. Default: 10000 milliseconds</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;log_read_batch&gt;</td>
<td>Determines how many DDS samples should be read and processed at once by the DDS sample log write thread. Default: 100 DDS samples</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;log_bookmark_period&gt;</td>
<td>DDS samples in the DDS sample log are identified by two attributes:</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>- The file ID</td>
<td></td>
</tr>
<tr>
<td></td>
<td>- The row ID (position within the file)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The read bookmark indicates the most recently processed DDS sample. This tag indicates how often (in milliseconds) the read bookmark is persisted into disk. Default: 1000 milliseconds</td>
<td></td>
</tr>
</tbody>
</table>

Enabling DDS sample logging in a persistence group is expensive. For every PRSTDataWriter, Persistence Service will create a write thread and an event thread that will be in charge of flushing the log files and storing the read bookmark. Therefore, DDS sample logging should be enabled only for the persistence groups where it is needed based on the potential presence of slow consumers and/or the expected data rate in the persistence group. Small data rates will likely not require a DDS sample log.
Part 9: RTI Monitoring Library

RTI Monitoring Library is a plug-in that enables Connext DDS applications to provide monitoring data. The monitoring data can be visualized with RTI Monitor, a separate GUI application that can run on the same host as Monitoring Library or on a different host.

Connext DDS notifies Monitoring Library every time an entity is created/deleted or a QoS is changed. Monitoring Library periodically queries the status of all Connext DDS entities. You can enable/disable monitoring by setting values in the DomainParticipant’s PropertyQosPolicy (programmatically or through an XML QoS profile).

This part of the User’s Manual includes:

- Using Monitoring Library in Your Application (Chapter 50 on page 1224)
- Configuring Monitoring Library (Chapter 51 on page 1234)
Chapter 50 Using Monitoring Library in Your Application

50.1 Enabling Monitoring

There are two ways to enable monitoring in your application:

- **50.1.1 Method 1**—Change the Participant QoS to Automatically Load the Dynamic Monitoring Library on the next page
- **50.1.2 Method 2**—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer and Explicitly Load the Monitoring Library on page 1226

**Notes:**

- The libraries that you will need for Monitoring are listed in the RTI Connext DDS Core Libraries Platform Notes.
- If your original application has made modifications to the ParticipantQos `resource_limits.type_code_max_serialized_length`, ParticipantQos `resource_limits.type_object_max_serialized_length`, or any of the transport's default settings to enable large type code or large data, refer to 50.3 What Monitoring Topics are Published? on page 1231 for additional QoS modifications that may be needed.
- *Monitoring Library* creates internal *DataWriters* to publish monitoring data by making modifications based on the default *DataWriter* QoS settings. If you have made changes to the default *DataWriter* QoS, especially if you have increased/decreased the initial or maximum DDS sample/instance values, *Monitoring Library* may have trouble creating *DataWriters* to publish monitoring data, or it may limit the number of statistics that you can publish through the internal monitoring writers. If this is true for your case, you may want to specify the `qos_library` and `qos_profile` that will be used to create these internal writers for publishing mon-
50.1.1 Method 1—Change the Participant QoS to Automatically Load the Dynamic Monitoring Library

Monitoring data, to avoid being impacted by default DataWriter QoS settings. See Configuring Monitoring Library (Chapter 51 on page 1234) for details.

50.1.1 Method 1—Change the Participant QoS to Automatically Load the Dynamic Monitoring Library

If all of the following are true, you can enable monitoring simply by changing your participant QoS (otherwise, use 50.1.2 Method 2—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer and Explicitly Load the Monitoring Library on the next page):

- Your application is linked to dynamic Connext DDS libraries (see 10.3.2 Mixing Static and Dynamic Libraries not Supported on page 682), or you are using Java or .Net, and
- You will run your application on a platform that is not VxWorks or INTEGRITY (which don't support this), and
- You are NOT linking in an additional monitoring library into your application at link time (you let the middleware load the monitoring library for you automatically as needed).

If you change the QoS in an XML file as shown below, you can enable/disble monitoring without recompiling. If you change the QoS in your source code, you may need to recompile every time you enable/disable monitoring.

If you need to change the participant QoS by hand, refer to the definition of Built-inQosLib::Generic.Monitoring.Common in <NDDSHOME>/resource/xml/BuiltinProfiles.documentationONLY.xml for the values you should set.

Example XML to enable monitoring:
50.1.2 Method 2—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer and Explicitly Load the Monitoring Library

If any of the following are true, you must change the Participant QoS to enable monitoring and explicitly load the correct version of Monitoring Library at compile time:

- Your application is linked to the static version of Connext DDS libraries. (See 10.3.2 Mixing Static and Dynamic Libraries not Supported on page 682.)
- You will run your application on a VxWorks or INTEGRITY platform.
- You want to explicitly link in the monitoring library (static or dynamic) into your application.

There are two ways to do this:

- **50.1.2.1 Method 2-A: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in Source Code on the next page:** Applies to most users who cannot use Method 1 and do not mind changing/recompiling source code every time you enable/disable monitoring, or whose system does not support setting environment variables programmatically. Participant QoS must be defined in source code with this approach.

- **50.1.2.2 Method 2-B: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in an Environment Variable on page 1230:** Applies to users who cannot use Method 1 and want to specify the create function pointer via an environment variable. This approach allows the Participant QoS to be defined in an XML file or in source code.
50.1.2.1 Method 2-A: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in Source Code

1. Modify your Connext DDS application based on the following examples.

**Traditional C++ Example:**

```cpp
#include "ndds/ndds_cpp.h"
#include "monitor/monitor_common.h"
extern "C" int publisher_main(int domainId, int sample_count)
{
    DDSDomainParticipant *participant = NULL;
    DDS_DomainParticipantQos participant_qos;

    /* Get default QoS */
    retcode = DDSTheParticipantFactory->get_default_participant_qos(
               participant_qos);
    if (retcode != DDS RETCODE_OK) {
        /*Error*/
    }

    /* This property indicates that the DomainParticipant has monitoring turned on. The property name MUST be "rti.monitor.library". The value can be anything.*/
    retcode = DDSPropertyQosPolicyHelper::add_property(
               participant_qos.property,
               "rti.monitor.library", "rtimonitoring", DDS_BOOLEAN_FALSE);
    if (retcode != DDS RETCODE_OK) {
        /*Error*/
    }

    /* The property name "rti.monitor.create_function" indicates the entry point for the monitoring library. The value MUST be the value of the function pointer of RTIDefaultMonitor_create */
    retcode = DDSPropertyQosPolicyHelper::add_pointer_property(
               participant_qos.property,
               "rti.monitor.create_function_ptr",
               (void *) RTIDefaultMonitor_create);
    if (retcode!= DDS RETCODE_OK) {
        /* Error */
    }

    /* Create DomainParticipant with participant_qos */
    participant = DDSTheParticipantFactory->create_participant(
                     domainId, participant_qos, NULL /* listener */,
                     DDS_STATUS_MASK_NONE);
    if (participant == NULL) {
        /* Error */
    }

    ...
```
50.1.2 Method 2—Change the Participant QoS to Specify the Monitoring Library Create Function Pointer

**Modern C++ Example:**

```cpp
#include "rti/rti.hpp" // include all the modern C++ API
#include "monitor/monitor_common.h" // for RTIDefaultMonitor_create
using rti::core::policy::Property;

// Get property policy from default DomainParticipantQos
auto participant_qos =
    dds::core::QosProvider::Default().participant_qos();
auto property_policy = participant_qos.policy<Property>();

// This property turns monitoring on
property_policy.set(Property::Entry("rti.monitor.library",
                                    "rtimonitoring"));

// This property specifies the entry point (function // pointer) for the monitoring library.
std::ostringstream monitor_function_to_str;
monitor_function_to_str <<
    reinterpret_cast<void*>(RTIDefaultMonitor_create);
property_policy.set(Property::Entry(
    "rti.monitor.create_function_ptr",
    monitor_function_to_str.str()));

participant_qos <<= property_policy;

// Create a DomainParticipant with Qos
dds::domain::DomainParticipant participant(0, participant_qos);
...
```

**C Example:**

```c
#include "ndds/ndds_c.h"
#include "monitor/monitor_common.h"
...
extern "C" int publisher_main(int domainId, int sample_count)
{
    DDS_DomainParticipantFactory *factory = NULL;
    struct DDS_DomainParticipantQos participantQos =
        DDS_DomainParticipantQos_INITIALIZER;

    DDS_DomainParticipant *participant = NULL;
    factory = DDS_DomainParticipantFactory_get_instance();
    if (factory == NULL) {
        /* error */
    }
}
```
if (DDS_DomainParticipantFactory_get_default_participant_qos(factory, &participantQos) != DDS_RETCODE_OK) {
    /* error */
} /* This property indicates that the DomainParticipant has monitoring turned on. The property name MUST be "rti.monitor.library". The value can be anything. */
if (DDS_PropertyQosPolicyHelper_add_property(&participantQos.property,
    "rti.monitor.library", "rtimonitoring",
    DDS_BOOLEAN_FALSE) != DDS_RETCODE_OK) {
    /* error */
} /* The property name "rti.monitor.create_function_ptr" indicates the entry point for the monitoring library. The value MUST be the value of the function pointer of RTIDefaultMonitor_create */
if (DDS_PropertyQosPolicyHelper_add_pointer_property(&participantQos.property,
    "rti.monitor.create_function_ptr", RTIDefaultMonitor_create)
    != DDS_RETCODE_OK) {
    /* error */
} /* create DomainParticipant with participantQos */
participant = DDS_DomainParticipantFactory_create_participant(
    factory, domainId, &participantQos,
    NULL /* listener */,
    DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    /* error */
}
DDS_DomainParticipantQos_finalize(&participantQos);
...

Note:
- In the above code, you may notice that valueBuffer is initialized to 17 characters. This is because a pointer (RTIDefaultMonitor_create) is at most 8 bytes (on a 64-bit system) and it takes two characters to represent a byte in hex. So the total size must be:

\[
(2 \times 8 \text{ characters}) + 1 \text{ null-termination character} = 17 \text{ characters}
\]

2. Link the Monitoring Library for your platform into your application at compile time (the Monitoring libraries are listed in the RTI Connext DDS Core Libraries Platform Notes).

The kind of monitoring library that you link into your application at compile time must be consistent with the kind of Connext DDS libraries that you are linking into your application (static/dynamic, release/debug version of the libraries).
On Windows systems: If you are linking a static monitoring library, you will also need to link in Psapi.lib at compile time.

50.1.2 Method 2-B: Change the Participant QoS by Specifying the Monitoring Library Create Function Pointer in an Environment Variable

This is similar to Method 2-A, but if you specify the function pointer value for rti.monitor.create_function_ptr in an environment variable that is set programmatically, you can specify your QoS either in an XML file or in source code. If you specify the QoS in an XML file, you can enable/disable monitoring without recompiling. If you change the QoS in your source code, you may need to recompile every time you enable/disable monitoring.

1. In XML, enable monitoring by setting the rti.monitor.create_function_ptr property to an environment variable. In our example, the variable is named RTIMONITORFUNCPtr.

   ```xml
   <participant_qos>
     <property>
       <value>
         <element>
           <name>rti.monitor.library</name>
           <value>rtimonitoring</value>
         </element>
         <element>
           <name>rti.monitor.create_function_ptr</name>
           <value>${RTIMONITORFUNCPtr}</value>
         </element>
       </value>
     </property>
   </participant_qos>
   ```

2. In the DDS application that links in the monitoring library, get the function pointer of RTIDefaultMonitor_create and write it to the same environment variable you named in Step 1 and create a DomainParticipant by using the XML profile specified in Step 1. (Setting of the environment variable must appear in the application before it creates the DomainParticipant using the profile from Step 1.)

Here is an example in C:

```c
#include <stdio.h>
#include <stdlib.h>
#include "monitor/monitor_common.h"
...
char putenvBuffer[34];
int putenvReturn;
putenvBuffer[0] = '0';
sprintf(putenvBuffer, "RTIMONITORFUNCPtr=%p",
      RTIDefaultMonitor_create);
putenvReturn = putenv(putenvBuffer);
if (putenvReturn) {
    printf("Error: couldn't set env variable for RTIMONITORFUNCPtr. "
           "error code: %d\n", putenvReturn );
```
50.2 How does Monitoring Library Work?

Monitoring Library works by creating DDS Topics that publish information about the other DDS entities contained in the same operating system process. The Topics can be created inside of the first DomainParticipant that enables the library (the default). Or they may be created in a separate DomainParticipant if the rti.monitor.config.new_participant_domain_id property is used. Use cases for this latter configuration include controlling the domain ID on which this information is exchanged (for example to ensure that this data does not interfere with production topics) as well as the ability to specify the QoS that is used for the DomainParticipant (through the rti.monitor.config.qos_library and rti-monitor.config.qos_profile properties). It may be desirable to specify the QoS for Monitoring Library's DomainParticipant if the information will be consumed on a different transport or simply to enable the feature but keep it as isolated from the production system as possible.

50.3 What Monitoring Topics are Published?

Two categories of predefined monitoring topics are sent out:

- **Descriptions** are published when an entity is created or deleted, or there are QoS changes (see Table 50.1 Descriptions (QoS and Other Static System Information)).
- **Entity Statistics** are published periodically (see Table 50.2 Entity Statistics (Statuses, Aggregated Statuses, CPU and Memory Usage)).
Table 50.1 Descriptions (QoS and Other Static System Information)

<table>
<thead>
<tr>
<th>Topic Name</th>
<th>Topic Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>rti/dds/monitoring/domainParticipantDescription</td>
<td>DomainParticipant QoS and other static information</td>
</tr>
<tr>
<td>rti/dds/monitoring/topicDescription</td>
<td>Topic QoS and other static information</td>
</tr>
<tr>
<td>rti/dds/monitoring/publisherDescription</td>
<td>Publisher QoS and other static information</td>
</tr>
<tr>
<td>rti/dds/monitoring/subscriberDescription</td>
<td>Subscriber QoS and other static information</td>
</tr>
<tr>
<td>rti/dds/monitoring/dataReaderDescription</td>
<td>DataReader QoS and other static information</td>
</tr>
<tr>
<td>rti/dds/monitoring/dataWriterDescription</td>
<td>DataWriter QoS and other static information</td>
</tr>
</tbody>
</table>

Table 50.2 Entity Statistics (Statuses, Aggregated Statuses, CPU and Memory Usage)

<table>
<thead>
<tr>
<th>Topic Name</th>
<th>Topic Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>rti/dds/monitoring/domainParticipantEntityStatistics</td>
<td>Number of entities discovered in the system, CPU and memory usage of the process</td>
</tr>
<tr>
<td>rti/dds/monitoring/dataReaderEntityStatistics</td>
<td>DataReader statuses</td>
</tr>
<tr>
<td>rti/dds/monitoring/dataWriterEntityStatistics</td>
<td>DataWriter statuses</td>
</tr>
<tr>
<td>rti/dds/monitoring/topicEntityStatistics</td>
<td>Topic statuses</td>
</tr>
<tr>
<td>rti/dds/monitoring/dataReaderEntityMatchedPublicationStatistics</td>
<td>DataReader statuses calculated on a per discovered matching writer basis</td>
</tr>
<tr>
<td>rti/dds/monitoring/dataWriterEntityMatchedSubscriptionStatistics</td>
<td>DataWriter statuses calculated on a per discovered matching reader basis</td>
</tr>
<tr>
<td>rti/dds/monitoring/dataWriterEntityMatchedSubscriptionWithLocatorStatistics</td>
<td>DataWriter statuses calculated on a per sending destination basis</td>
</tr>
</tbody>
</table>

All monitoring data are sent out using specially created DataWriters with the above topics.

You can configure some aspects of Monitoring Library’s behavior, such as which monitoring topics to turn on, which user topics to monitor, how often to publish the statistics topics, and whether to publish monitoring data using (a) the participant created in the user’s application that has monitoring turned on or (b) a separate participant created just for publishing monitoring data. See Configuring Monitoring Library (Chapter 51 on page 1234).

50.4 Enabling Support for Large Type-Code (Optional)

Some monitoring topics have large type-code (larger than the default maximum type code serialized size setting). If you use Monitor to display all the monitoring data, it already has all the monitoring types built-
in and therefore it uses the default maximum type-code serialized size in the Connext DDS application and there is no problem. However, if you are using any other tools to display monitoring data (such as RTI Spreadsheet Add-in for Microsoft Excel, rtiddspy, or writing your own application to subscribe to monitoring data), or if your user data-type has large type-code, you may need to increase the maximum type-object serialized size setting in the DomainParticipantResourceLimitsQosPolicy.

50.5 Troubleshooting Monitoring

50.5.1 Buffer Allocation Error

Monitoring Library obtains the default DataWriter QoS from the Connext DDS application’s DomainParticipant. If the application has changed the default QoS Profile, either through application code or in an XML file, Monitoring Library will use this new default QoS. In specific scenarios, the new default QoS may cause your Connext DDS application to run out of memory and report error messages similar to these:

```plaintext
REDAFastBufferPool_growEmptyPoolEA: !allocate buffer of 1210632000 bytes
[D0012|ENABLE]REDAFastBufferPool_newWithNotification:!create fast buffer pool buffers
[D0012|ENABLE]PRESTypePluginDefaultEndpointData_createWriterPool:!create writer buffer pool
[D0012|ENABLE]WriterHistorySessionManager_new:!create new Allocator
[D0012|ENABLE]WriterHistoryMemoryPlugin_createHistory:!create sessionManager
[D0012|ENABLE]PRESWriterHistoryDriver_new:!create _whHnd
[D0012|ENABLE]PRESpsService_enableLocalEndpointWithCursor:!create WriterHistoryDriver
[D0012|ENABLE]PRESpsService_enableAllLocalEndpointsInGroupWithCursor!enable endpoint
[D0012|ENABLE]PRESpsService_enableGroupWithCursor:!enableAllLocalEndpointsInGroupWithCursor
[D0012|ENABLE]PRESpsService_enableGroup:!enableGroupWithCursor
[D0012|ENABLE]RTIDefaultMonitorPublisher_enableEntitiesAndStartThreadI:!create enable publisher
[D0012|ENABLE]RTIDefaultMonitorPublisher_onEventNotify:!create enable entities
```

To resolve this problem, either:

- Configure Monitoring Library to use a non-default QoS Profile. For details, see Configuring Monitoring Library (Chapter 51 on page 1234).
- Change the default QoS to have a lower value for DataWriter’s initial_samples; this field is part of the ResourceLimitsQosPolicy.
Chapter 51 Configuring Monitoring Library

You can control some aspects of Monitoring Library’s behavior by setting the PropertyQosPolicy of the DomainParticipant, either via an XML QoS profile or in your application’s code prior to creating the DomainParticipant.

Two example QoS profiles are provided in
<path to examples>/connext_dds/qos/MONITORING_LIBRARY_QOS_PROFILES.xml
(see Paths Mentioned in Documentation on page 1):

- CustomerExampleMonitoringLibrary::CustomerExampleMonitoringProfile
  This is an example of how to enable Monitoring Library for your applications. It can be used as a guide to enabling Monitoring Library quickly in your applications.

- RTIMonitoringQosLibrary::RTIMonitoringQosProfile
  This profile documents the QoS used by Monitoring Library. It can also be used as a starting point if you want to tune QoS for Monitoring Library (normally not necessary). Use cases for this include customizing DomainParticipant QoS (often the transports) to accommodate preferences or environment. This same profile can also be used to subscribe to the Monitoring Library Topics. This is useful in situations where the Monitoring Library information can be used directly by system components or it is not possible to use the RTI Monitor tool.


See the qos_library on page 1236 and qos_profile on page 1236 properties in Table 51.1 Configuration Properties for Monitoring Library for further information on when to use the example profiles in MONITORING_LIBRARY_QOS_PROFILES.xml.
Table 51.1 Configuration Properties for Monitoring Library lists the configuration properties that you can set for Monitoring Library. These properties are immutable; they cannot be changed after the DomainParticipant is created.

### Table 51.1 Configuration Properties for Monitoring Library

<table>
<thead>
<tr>
<th>Property Name (all must be prepended with “rti.monitor.config.”)</th>
<th>Property Value</th>
</tr>
</thead>
</table>
| get_process_statistics | This boolean value specifies whether or not Monitoring Library should collect CPU and memory usage statistics for the process in the topic rti.dds/monitoring/domainParticipantDescription.  
This property is only applicable for platforms that support obtaining CPU and memory usage from monitoring data. 
CPU and memory usage is not available on these platforms: VxWorks, INTEGRITY. 
CPU usage is reported in terms of time spent since the process has been started. It can be longer than the actual running time of the process on a multi-core machine. 
Default: true if unspecified |
| new_participant_domain_id | To create a separate participant that will be used to publish monitoring information in the application, set this to the domain ID that you want to use for the newly created participant. 
The new participant is created with the default Qos (for example, that defined in USER_QOS_PROFILES.xml), unless the qos_library on the next page and qos_profile on the next page properties are set. 
Default: Not set (means you want to reuse the participant in your application that has monitoring turned on to publish statistics information for that participant) |
| publish_period | Period of time to sample and publish all monitoring topics, in units of seconds. 
Default: 5 if unspecified |
| publish_thread_priority | Priority of the thread used to sample and publish monitoring data. 
This value is architecture dependent. 
Default if unspecified: same as the default used in Connext DDS for the event thread: 
Windows systems: -2 
Linux systems: -999999 (meaning use OS-default priority) |
| publish_thread_stacksize | Stack size used for the thread that samples and publishes monitoring data. This value is architecture dependent. 
Default if unspecified: same as the default used in Connext DDS for the event thread: 
Windows systems: -1 (meaning use the default size for the executable). 
Linux systems: -1 (meaning use OS’s default value). |
### Table 51.1 Configuration Properties for Monitoring Library

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(all must be prepended with</td>
<td></td>
</tr>
<tr>
<td>“rti.monitor.config.”)</td>
<td></td>
</tr>
<tr>
<td>publish_thread_options</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Describes the type of thread.</td>
</tr>
<tr>
<td></td>
<td>Supported values (may be combined with by OR’ing with ↑ as seen in the default below):</td>
</tr>
<tr>
<td></td>
<td>• FLOATING_POINT: Code executed within the thread may perform floating point operations</td>
</tr>
<tr>
<td></td>
<td>• STDIO: Code executed within the thread may access standard</td>
</tr>
<tr>
<td></td>
<td>• REALTIME_PRIORITY: The thread will be scheduled on a real-time basis</td>
</tr>
<tr>
<td></td>
<td>• PRIORITY_ENFORCE: Strictly enforce this thread’s priority</td>
</tr>
<tr>
<td></td>
<td>Default: FLOATING_POINT</td>
</tr>
<tr>
<td>qos_library</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the QoS library that will be used to create the monitoring library DomainParticipant, Publisher, and DataWriters.</td>
</tr>
<tr>
<td></td>
<td>Default: Not set. If you don’t set this property, the entities are created with the following QoS values:</td>
</tr>
<tr>
<td></td>
<td>• The DomainParticipant uses the default DomainParticipantQos (for example, that defined in USER_QOS_PROFILES.xml).</td>
</tr>
<tr>
<td></td>
<td>• The Publisher and the DataWriters use a specific QoS configuration that can be found in the library RTIMonitoringQosLibrary in &lt;path to examples&gt;/connex_dds/qos/MONITORING_LIBRARY_QOS_PROFILES.xml. (Note that the Publisher and DataWriters use the values reproduced in this .xml file, but modifying the file has no effect; it is for reference only.)</td>
</tr>
<tr>
<td>qos_profile</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Specifies the name of the QoS profile that will be used to create the monitoring library DomainParticipant, Publisher, and DataWriters.</td>
</tr>
<tr>
<td></td>
<td>Default: Not set. If you don’t set this property, the entities are created with the following QoS values:</td>
</tr>
<tr>
<td></td>
<td>• The DomainParticipant uses the default DomainParticipantQos (for example, that defined in USER_QOS_PROFILES.xml).</td>
</tr>
<tr>
<td></td>
<td>• The Publisher and the DataWriters use a specific QoS configuration that can be found in the library RTIMonitoringQosLibrary in &lt;path to examples&gt;/connex_dds/qos/MONITORING_LIBRARY_QOS_PROFILES.xml. (Note that the Publisher and DataWriters use the values reproduced in this .xml file, but modifying the file has no effect; it is for reference only.)</td>
</tr>
<tr>
<td>reset_status_change_counts</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Monitoring Library obtains all statuses of all entities in the Connext DDS application. This boolean value controls whether or not the change counts in those statuses are reset by Monitoring Library.</td>
</tr>
<tr>
<td></td>
<td>If set to true, the change counts are reset each time Monitoring Library is done accessing them.</td>
</tr>
<tr>
<td></td>
<td>If set to false, the change counts truly reflect what users will see in their application and are unaffected by the access of the monitoring library.</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
</tr>
</tbody>
</table>
### Table 51.1 Configuration Properties for Monitoring Library

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip_monitor_entities</td>
<td>This boolean value controls whether or not the entities created internally by Monitoring Library should be included in the entity counts published by the participant entity statistics topic. If set to true, the internal monitoring entities will not be included in the count. (Thirteen internal writers are created by the monitoring library by default.) Default: true</td>
</tr>
</tbody>
</table>
| skip_participant_properties | If set to true, DomainParticipantPropertyQosPolicy name and value pairs will not be sent out through the DomainParticipantDescriptionTopic. This is necessary if you are linking with Monitoring Library and any of these conditions occur:  
  - The PropertyQosPolicy of a DomainParticipant has more than 64 properties.  
  - Any of the properties in PropertyQosPolicy of a DomainParticipant has a name longer than 127 characters or a value longer than 511 characters. Default: false if unspecified |
| skip_reader_properties | If set to true, DataReader PropertyQosPolicy name and value pairs will not be sent out through the DataReaderDescriptionTopic. This is necessary if you are linking with Monitoring Library and any of these conditions occur:  
  - The PropertyQosPolicy of a DataReader has more than 64 properties.  
  - Any of the properties in PropertyQosPolicy of a DataReader has a name longer than 127 characters or a value longer than 511 characters. Default: false if unspecified |
| skip_writer_properties | If set to true, DataWriter PropertyQosPolicy name and value pairs will not be sent out through the DataWriterDescriptionTopic. This is necessary if you are linking with Monitoring Library and any of these conditions occur:  
  - The PropertyQosPolicy of a DataWriter has more than 64 properties.  
  - Any of the properties in PropertyQosPolicy of a DataWriter has a name longer than 127 characters or a value longer than 511 characters. Default: false if unspecified |
| topics            | Filter for monitoring topics, with regular expression matching syntax as specified in the Connext DDS documentation (similar to the POSIX fnmatch syntax). For example, if you only want to send description topics and the entity statistics topics, but NOT the matching statistics topics, you can specify **Description, EntityStatistics**. Default: * if unspecified |
| usertopics         | Filter for user topics, with regular expression matching syntax as specified in the Connext DDS documentation (similar to the POSIX fnmatch syntax). For example, if you only want to send monitoring information for reader/writer/topic entities for topics that start with Foo or Bar, you can specify *Foo,Bar*. Default: * if unspecified |
### Table 51.1 Configuration Properties for Monitoring Library

<table>
<thead>
<tr>
<th>Property Name</th>
<th>Property Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>verbosity</td>
<td>Sets the verbosity on the monitoring library for debugging purposes (does not affect the topic/data that is sent out).</td>
</tr>
<tr>
<td></td>
<td>-1: Silent</td>
</tr>
<tr>
<td></td>
<td>0: Exceptions only</td>
</tr>
<tr>
<td></td>
<td>1: Warnings</td>
</tr>
<tr>
<td></td>
<td>2 and up: Higher verbosity level</td>
</tr>
<tr>
<td></td>
<td>Default: 1 if unspecified</td>
</tr>
<tr>
<td>writer_pool_buffer_max_size</td>
<td>Controls the threshold at which dynamic memory allocation is used, expressed as a number of bytes.</td>
</tr>
<tr>
<td></td>
<td>If the serialized size of the data to be sent is smaller than this size, a pre-allocated writer buffer pool is used to obtain the memory.</td>
</tr>
<tr>
<td></td>
<td>If the serialized size of the data is larger than this value, the memory is allocated dynamically.</td>
</tr>
<tr>
<td></td>
<td>This setting can be used to control memory consumption of the monitoring library, at the cost of performance, when the maximum serialized size of the data type is large (which is the case for some description topics' data types) or if you have several participants on the same machine.</td>
</tr>
<tr>
<td></td>
<td>The default setting is -1, meaning memory is always obtained from the writer buffer pool, whose size is determined by the maximum serialized size.</td>
</tr>
</tbody>
</table>
Part 10: RTI Distributed Logger

RTI Distributed Logger is a library that enables applications to publish their log messages to Connext DDS. The log message data can be visualized with RTI Monitor and RTI Admin Console. Since the data is provided in a Topic, you can also use rtiddspy or even write your own visualization tool.

Distributed Logger can also send Connext DDS errors, warnings and other internal messages from its own builtin logging system as a DDS Topic. In fact, Distributed Logger provides a remote command topic so that its behavior can be remotely controlled at run time.

This part of the User’s Manual includes:

- Using Distributed Logger in a Connext DDS Application (Chapter 52 on page 1240)
- Enabling Distributed Logger in RTI Services (Chapter 53 on page 1250)
Chapter 52 Using Distributed Logger in a Connext DDS Application

There are two ways to use Distributed Logger: directly through its API or by attaching it to an existing logging framework as an ‘appender’ or a ‘handler.’ Using the API directly is straightforward, but keep in mind that Distributed Logger is not intended to be a full-featured logging library. Rather, it is primarily intended to be integrated into third-party logging infrastructures.

The libraries that you will need for Distributed Logger are listed in 52.1 Distributed Logger Libraries below.

Distributed Logger comes with third-party integrations for the open-source project log4j (http://logging.apache.org/log4j/) as well as Java’s built-in logging library (java.util.logging). Please see 52.3 Examples on the next page for examples that illustrate these integrations.

Distributed Logger captures and forwards Connext DDS internal information, warning, and error messages using a DDS topic. It monitors these messages using the same mechanism as user log messages.

These Connext DDS log messages are sent over DDS automatically as soon as you initialize Distributed Logger (by calling RTI_DL_DistLogger_getInstance() in C or C++, or Logger.getLogger(...) in Java; see the API Reference HTML documentation for details).

52.1 Distributed Logger Libraries

Table 52.1 Required Libraries lists the additional libraries you will need in order to use Distributed Logger.
Table 52.1 Required Libraries

<table>
<thead>
<tr>
<th>Platform</th>
<th>Language</th>
<th>Static Release</th>
<th>Static Debug</th>
<th>Dynamic Release</th>
<th>Dynamic Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux®</td>
<td>C</td>
<td>librtidlcz.a</td>
<td>librtidlczd.a</td>
<td>librtidl.so</td>
<td>librtidlcd.so</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>librtidlcz.a</td>
<td>librtidlczd.a</td>
<td>librtidlc.so</td>
<td>librtidlcd.so</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>N/A</td>
<td>N/A</td>
<td>distlog.jar</td>
<td>distlogd.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distlogdatamodel.jar</td>
<td>distlogdatamodeld.jar</td>
</tr>
<tr>
<td>QNX</td>
<td>C</td>
<td>librtidlcz.a</td>
<td>librtidlczd.a</td>
<td>librtidl.so</td>
<td>librtidlcd.so</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>librtidlcz.a</td>
<td>librtidlczd.a</td>
<td>librtidlc.so</td>
<td>librtidlcd.so</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>N/A</td>
<td>N/A</td>
<td>distlog.jar</td>
<td>distlogd.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distlogdatamodel.jar</td>
<td>distlogdatamodeld.jar</td>
</tr>
<tr>
<td>VxWorks™</td>
<td>C</td>
<td>librtidlcz.a</td>
<td>librtidlczd.a</td>
<td>librtidl.so</td>
<td>librtidlcd.so</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>librtidlcz.a</td>
<td>librtidlczd.a</td>
<td>librtidlc.so</td>
<td>librtidlcd.so</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>N/A</td>
<td>N/A</td>
<td>distlog.jar</td>
<td>distlogd.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distlogdatamodel.jar</td>
<td>distlogdatamodeld.jar</td>
</tr>
<tr>
<td>Windows®</td>
<td>C</td>
<td>rtidlcz.lib</td>
<td>rtidlczd.lib</td>
<td>rtidlc.dll</td>
<td>rtidlcd.dll</td>
</tr>
<tr>
<td></td>
<td>C++</td>
<td>rtidlcz.lib</td>
<td>rtidlczd.lib</td>
<td>rtidlc.dll</td>
<td>rtidlcpp.dll</td>
</tr>
<tr>
<td></td>
<td>Java</td>
<td>N/A</td>
<td>N/A</td>
<td>distlog.jar</td>
<td>distlogd.jar</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>distlogdatamodel.jar</td>
<td>distlogdatamodeld.jar</td>
</tr>
</tbody>
</table>

52.2 Using the API Directly

Details on using the Distributed Logger APIs are provided in the API Reference HTML documentation: 
<NDDSHOME>/doc/api/connex_dds/distributed_logger/<language>. Start by opening index.html. (See Paths Mentioned in Documentation on page 1)

If you plan to use the Distributed Logger’s API directly, please be aware of the following notes. To configure the options, create an options object and update its fields. Once your updates are complete, set the options on Distributed Logger. It is important that this be done before Distributed Logger is instantiated. Distributed Logger acts as a singleton and there is no way to change the options after it has been created.

When your application is ready to exit, use the ‘delete’ method. This will delete all Entities and threads associated with Distributed Logger.

52.3 Examples

Distributed Logger includes several examples in <path to examples>/distributed_logger (see Paths Mentioned in Documentation on page 1):
- **c/hello_distributed_logger**
  
  This is a simple example of how to use the API directly and does not publish or subscribe to any Topics except the ones related to Distributed Logger.

- **c++/hello_distributed_logger**
  
  This is a simple example of how to use the API directly and does not publish or subscribe any Topics except the ones related to Distributed Logger.

- **java/hello_direct_usage**
  
  This is a simple example of how to use the API directly and does not publish or subscribe any Topics except the ones related to Distributed Logger.

- **java/hello_file_logger**
  
  This example shows how an application can use the information provided by Distributed Logger. As the name suggests, this example subscribes to log messages and writes them to a file. Multiple DDS domains can be subscribed to simultaneously if desired. The example is meant to strike a balance between simplicity and function. Certainly more features could be added to make it a production-ready application but that would obscure the goal of the example.

- **java/hello_java_util_logging**
  
  In this example, all System.\{out/err\} invocations are replaced with Java logging library equivalents. It adds Distributed Logger through a configuration file.

- **java/hello_log4j_logging**
  
  In this example, all System.\{out/err\} invocations are replaced with log4j library equivalents. It adds Distributed Logger through a configuration file.

Each example has a READ_ME.txt file which explains how to build and run it.

### 52.4 Data Type Resource

You can find the data types used by Distributed Logger in `<NDDSHOME>/resource/idl/distog.idl`. (See Paths Mentioned in Documentation on page 1.)

If you want to generate code and interact with Distributed Logger through Topics, you can use this file to do so. You will need to provide extra command-line arguments to RTI Code Generator (`rtiddsgen`). (This allows us to accommodate multiple language bindings within the same file. As a consequence, we’ve used preprocessor definitions to achieve this functionality.) The command-line options which must be added to `rtiddsgen` are as follows:
### 52.5 Distributed Logger Topics

**Distributed Logger** uses four Topics to publish log messages, state, and command responses and one topic to subscribe to command requests. These are detailed in **Table 52.3 Topics Used by Distributed Logger**.

#### Table 52.3 Topics Used by Distributed Logger

<table>
<thead>
<tr>
<th>Topic</th>
<th>Type Name</th>
<th>Quality of Service</th>
</tr>
</thead>
</table>

If you plan to use the generated code in your application (to subscribe to log messages, for instance) be aware that the type names used might not match the default ones. *Do not* use the generated type names obtained when calling `get_type_name()` or found in `distlogSupport.h`. Use the variables in **Table 52.2 Registration Names for each Distributed Logger Type** instead.

#### Table 52.2 Registration Names for each Distributed Logger Type

<table>
<thead>
<tr>
<th>Type</th>
<th>Registered Typename</th>
<th>Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Message</td>
<td><code>com::rti::dl::LogMessage</code></td>
<td>C/C++: <code>RTI_DL_LOG_MESSAGE_TYPE_NAME</code> Java: <code>LOG_MESSAGE_TYPE_NAME.VALUE</code></td>
</tr>
<tr>
<td>Administration State</td>
<td><code>com::rti::dl::admin::State</code></td>
<td>C/C++: <code>RTI_DL_STATE_TYPE_NAME</code> Java: <code>STATE_TYPE_NAME.VALUE</code></td>
</tr>
<tr>
<td>Administration Command Request</td>
<td><code>com::rti::dl::admin::CommandRequest</code></td>
<td>C/C++: <code>RTI_DL_COMMAND_REQUEST_TYPE_NAME</code> Java: <code>COMMAND_REQUEST_TYPE_NAME.VALUE</code></td>
</tr>
<tr>
<td>Administration Command Response</td>
<td><code>com::rti::dl::admin::CommandResponse</code></td>
<td>C/C++: <code>RTI_DL_COMMAND_RESPONSE_TYPE_NAME</code> Java: <code>COMMAND_RESPONSE_TYPE_NAME.VALUE</code></td>
</tr>
</tbody>
</table>

For instance, to subscribe to log messages in C you will need to do the following:

```c
recode = RTI_DL_LogMessageTypeSupport_register_type(
    participant, RTI_DL_LOG_MESSAGE_TYPE_NAME);
```

For C or C++: `-DLANGUAGE_C`

For Java: `-DLANGUAGE_JAVA`

For .Net: `-DLANGUAGE_DOTNET`
Table 52.3 Topics Used by Distributed Logger

<table>
<thead>
<tr>
<th>Topic Path</th>
<th>Type Path</th>
<th>Reliability</th>
</tr>
</thead>
<tbody>
<tr>
<td>rti/distlog</td>
<td>com::rti::dl::LogMessage</td>
<td>Reliable</td>
</tr>
<tr>
<td>rti/distlog/administration/state</td>
<td>com::rti::dl::admin::State</td>
<td>Reliable, Transient, Local</td>
</tr>
<tr>
<td>rti/distlog/administration/command_request</td>
<td>com::rti::dl::admin::CommandRequest</td>
<td>Reliable</td>
</tr>
<tr>
<td>rti/distlog/administration/command_response</td>
<td>com::rti::dl::admin::CommandResponse</td>
<td>Reliable</td>
</tr>
</tbody>
</table>

52.6 Distributed Logger IDL

The IDL describing the types used for Topics created by Distributed Logger are in `<NDDSHOME>/resource/idl/distlog.idl`. (See Paths Mentioned in Documentation on page 1.) You can use this IDL to create custom applications that use the data provided by Distributed Logger and/or to remotely control any Distributed Logger instances that are running in your system. The IDL has been designed to take advantage of the latest type-support features in Connext DDS.

52.7 Viewing Log Messages

One way to see the messages from Distributed Logger is to use RTI Monitor.
Figure 52.1: Viewing Log Messages with RTI Monitor

Other ways to see the log messages include using riddsspy or writing your own visualization tool. If you want to write your own application that interacts with Distributed Logger, you can find the IDL in <NDDSHOME>/resource/idl/distlog.idl. (See Paths Mentioned in Documentation on page 1.)

52.8 Logging Levels

Log levels in Distributed Logger are organized as shown in Table 52.4 Mapping between Distributed Logger and Connext DDS Builtin Logging System (ordered by importance). This table also shows the mapping between logging levels in the Connext DDS builtin logging system and Distributed Logger.

**Table 52.4 Mapping between Distributed Logger and Connext DDS Builtin Logging System**

<table>
<thead>
<tr>
<th>Connext DDS Builtin Logging System Log Level</th>
<th>Distributed Logger Log Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>NDDS_CONFIG_LOG_FATAL_ERROR</td>
<td>RTI_DL_FATAL_LEVEL</td>
</tr>
<tr>
<td>NDDS_CONFIG_LOG_LEVEL_ERROR</td>
<td>RTI_DL_ERROR_LEVEL</td>
</tr>
<tr>
<td>NDDS_CONFIG_LOG_LEVEL_WARNING</td>
<td>RTI_DL_WARNING_LEVEL</td>
</tr>
<tr>
<td>NDDS_CONFIG_LOG_LEVEL_STATUS_LOCAL</td>
<td>RTI_DL_NOTICE_LEVEL</td>
</tr>
</tbody>
</table>
52.9 Distributed Logger Quality of Service Settings

To ensure that Distributed Logger works correctly with other RTI tools, some QoS settings are hard-coded and cannot be modified by customized profiles. Table 52.5 QoS Values Used by Distributed Logger lists the QoS values that are set in Distributed Logger. Values in bold are hard-coded; therefore even if they appear in an XML profile, they remain as noted in the table.

Table 52.5 QoS Values Used by Distributed Logger

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscriber</td>
<td>Presentation.access_scope</td>
<td>PRES_INSTANCE_PRESENTATION_QOS</td>
</tr>
<tr>
<td></td>
<td>Presentation.coherent_access</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>Presentation.ordered_access</td>
<td>false</td>
</tr>
<tr>
<td>Publisher</td>
<td>Presentation.access_scope</td>
<td>PRES_INSTANCE_PRESENTATION_QOS</td>
</tr>
<tr>
<td></td>
<td>Presentation.coherent_access</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>Presentation.ordered_access</td>
<td>false</td>
</tr>
<tr>
<td>Log Message Topic</td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>Durability.kind</td>
<td>DDS_TRANSIENT_LOCAL_DURABILITY_QOS</td>
</tr>
<tr>
<td>Administration State Topic</td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>Durability.kind</td>
<td>DDS_TRANSIENT_LOCAL_DURABILITY_QOS</td>
</tr>
<tr>
<td>Administration Command Request Topic</td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
<tr>
<td>Administration Command Response Topic</td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
</tbody>
</table>
### Table 52.5 QoS Values Used by Distributed Logger

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log Message DataWriter</td>
<td>Ownership.kind</td>
<td>DDS_SHARED_ownership_QOS</td>
</tr>
<tr>
<td></td>
<td>Latency_budget.duration.sec</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Latency_budget.duration.nanosec</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Liveliness.kind</td>
<td>DDS_AUTOMATIC_Liveliness_QOS</td>
</tr>
<tr>
<td></td>
<td>Destination_order.kind</td>
<td>DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</td>
</tr>
<tr>
<td></td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>Durability.kind</td>
<td>DDS_TRANSIENT_LOCAL_DURABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.kind</td>
<td>DDS_KEEP_LAST_HISTORY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.depth</td>
<td>10</td>
</tr>
<tr>
<td>Administration State DataWriter</td>
<td>Ownership.kind</td>
<td>DDS_SHARED_ownership_QOS</td>
</tr>
<tr>
<td></td>
<td>Latency_budget.duration.sec</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Latency_budget.duration.nanosec</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Liveliness.kind</td>
<td>DDS_AUTOMATIC_Liveliness_QOS</td>
</tr>
<tr>
<td></td>
<td>Destination_order.kind</td>
<td>DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</td>
</tr>
<tr>
<td></td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>Durability.kind</td>
<td>DDS_TRANSIENT_LOCAL_DURABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.kind</td>
<td>DDS_KEEP_LAST_HISTORY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.depth</td>
<td>1</td>
</tr>
<tr>
<td>Administration Command Response DataWriter</td>
<td>Ownership.kind</td>
<td>DDS_SHARED_ownership_QOS</td>
</tr>
<tr>
<td></td>
<td>Latency_budget.duration.sec</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Latency_budget.duration.nanosec</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Liveliness.kind</td>
<td>DDS_AUTOMATIC_Liveliness_QOS</td>
</tr>
<tr>
<td></td>
<td>Destination_order.kind</td>
<td>DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</td>
</tr>
<tr>
<td></td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.kind</td>
<td>DDS_KEEP_LAST_HISTORY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.depth</td>
<td>10</td>
</tr>
</tbody>
</table>
Table 52.5 QoS Values Used by Distributed Logger

<table>
<thead>
<tr>
<th>Entity</th>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Administration Command</td>
<td>Ownership.kind</td>
<td>DDS_SHARED_ownership_QOS</td>
</tr>
<tr>
<td>Request DataReader</td>
<td>Latency_budget.duration.sec</td>
<td>DDS_DURATION_INFINITE_SEC</td>
</tr>
<tr>
<td></td>
<td>Latency_budget.duration.nanosec</td>
<td>DDS_DURATION_INFINITE_NSEC</td>
</tr>
<tr>
<td></td>
<td>Deadline.period.sec</td>
<td>DDS_DURATION_INFINITE_SEC</td>
</tr>
<tr>
<td></td>
<td>Deadline.period.nanosec</td>
<td>DDS_DURATION_INFINITE_NSEC</td>
</tr>
<tr>
<td></td>
<td>Liveliness.kind</td>
<td>DDS_AUTOMATIC_LIVELINESS_QOS</td>
</tr>
<tr>
<td></td>
<td>Destination_order.kind</td>
<td>DDS_BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS</td>
</tr>
<tr>
<td></td>
<td>Reliability.kind</td>
<td>DDS_RELIABLE_RELIABILITY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.kind</td>
<td>DDS_KEEP_LAST_HISTORY_QOS</td>
</tr>
<tr>
<td></td>
<td>History.depth</td>
<td>10</td>
</tr>
</tbody>
</table>

52.10 Troubleshooting

52.10.1 Message Losses

In some cases, some of the messages logged with Distributed Logger can be lost, especially when the log message generation rate is high.

You can detect losses by inspecting the LogMessage.messageId field when subscribing to log messages. If the difference between two consecutive log messages coming from the same LogMessage.hostAndAppId is greater than 1, then some messages were lost.

Messages can be lost for two reasons:

- The queue that Distributed Logger uses to temporarily store log messages from the application until they can be written to Connext DDS is full.
- A log message published by Distributed Logger is replaced on the Connext DDS DataWriter queue before the DataReaders have a chance to receive it.

To minimize losses in the Distributed Logger queue, increase the queue size by using the API RTI_DLOptions::setQueueSize.

To minimize losses in the DataWriter that publishes the log messages, increase the number of messages that the DataWriter is caching for the process where Distributed Logger is running. To increase this number, configure writer_qos.history.depth. You can change the history depth in the QoS profile that you
use to configure *Distributed Logger*, by invoking the APIs `RTI_DLOptions::setQosLibrary` and `RTI_DLOptions::setQosProfile`.

### 52.10.2 Logger Device not Working

If you enable *Distributed Logger*, any previously created logger device will not be used. This is because you cannot have more than one logger device installed. When you enable *Distributed Logger*, it overwrites any previously created logger device.

Suppose *Distributed Logger* is enabled, and you are using a profile that configures *Connext DDS* to print the log messages to a file, such as:

```xml
<qos_profile name="..." is_default_qos="true">
  <participant_factory_qos>
    <logging>
      <output_file>/path/to/log/file/log.txt</output_file>
    </logging>
  </participant_factory_qos>
  ...
</qos_profile>
```

In this case, messages will be printed by *Distributed Logger*, they will not be sent to the log file.

If you want messages to be printed to the log file, you need to disable *Distributed Logger* first.
Chapter 53 Enabling Distributed Logger in RTI Services

Many RTI components provide integrated support for Distributed Logger (check the component’s Release Notes) and include the Distributed Logger library in their distribution. To enable Distributed Logger in these components, modify their XML configuration file. In the <administration> section, add the <distributed_logger> tag as shown in this example:

```xml
<persistence_service name="default">
  <administration>
    <domain_id>10</domain_id>
    <distributed_logger>
      <enabled>true</enabled>
      <filter_level>DEBUG</filter_level>
      <queue_size>2048</queue_size>
      <thread>
        <priority>
          THREAD_PRIORITY BELOW NORMAL
        </priority>
        <stack_size>8192</stack_size>
        <cpu_list>
          <element>0</element>
          <element>1</element>
        </cpu_list>
        <cpu_rotation>
          THREAD_SETTINGS_CPU_NO_ROTATION
        </cpu_rotation>
      </thread>
    </distributed_logger>
  </administration>
...  
</persistence_service>
```

The tags supported within the <distributed_logger> tag are described in Table 53.1 Distributed Logger Tags.
## Table 53.1 Distributed Logger Tags

<table>
<thead>
<tr>
<th>Tags within &lt;distributed_logger&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enabled&gt;</td>
<td>Controls whether or not Distributed Logger should be enabled at start up. This field is required. Allowed values: TRUE or FALSE</td>
<td>1 (required)</td>
</tr>
<tr>
<td>&lt;filter_level&gt;</td>
<td>The filter level for the log messages to be sent. Distributed Logger uses the filter level to discard log messages before they can be sent from the application/service. This is the minimum log level that will be sent out over the network. For example, when using the NOTICE level, any INFO, DEBUG and TRACE-level log messages will be filtered out and not sent from the application/service to Connext DDS. <strong>See important information in 53.1 Relationship Between Service Verbosity and Filter Level on page 1253.</strong> Can be set to these values:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SILENT</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• FATAL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• SEVERE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• ERROR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• WARNING</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• NOTICE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• INFO</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• DEBUG</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• TRACE (most verbose level, default)</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;queue_size&gt;</td>
<td>The size of an internal message queue used to store log messages before they are written to DDS. Default, 128 log messages.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;echo_to_stdout&gt;</td>
<td>Controls whether or not Distributed Logger should echo log messages to standard output (true) or not (false). Allowed values: TRUE or FALSE Default: TRUE</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;log_infrastructure_messages&gt;</td>
<td>Controls whether or not Distributed Logger should log infrastructure messages Allowed values: TRUE or FALSE Default: TRUE</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;thread&gt;</td>
<td>See Table 53.2 Distributed Logger Thread Tags.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
### Table 53.2 Distributed Logger Thread Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;distributed_logger&gt;/ &lt;thread&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;cpu_list&gt;</code></td>
<td>Each <code>&lt;element&gt;</code> specifies a processor on which the Distributed Logger thread may run. <code>&lt;cpu_list&gt;</code> value <code>&lt;element&gt;</code> <code>&lt;cpu_list&gt;</code></td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>Only applies to platforms that support controlling CPU core affinity (see the RTI Connext DDS Core Libraries Platform Notes).</td>
<td></td>
</tr>
<tr>
<td><code>&lt;cpu_rotation&gt;</code></td>
<td>Determines how the CPUs in <code>&lt;cpu_list&gt;</code> will be used by the Distributed Logger thread. The value can be either:</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
|                                            | - THREAD_SETTINGS_CPU_NO_ROTATION  
  The thread can run on any listed processor, as determined by OS scheduling. | |
|                                            | - THREAD_SETTINGS_CPU_RR_ROTATION  
  The thread will be assigned a CPU from the list in round-robin order. | |
|                                            | Only applies to platforms that support controlling CPU core affinity (see the RTI Connext DDS Core Libraries Platform Notes). | |
| `<mask>`                                  | A collection of flags used to configure threads of execution. Not all of these options may be relevant for all operating systems. May include these bits: | 0 or 1 |
|                                            | - STDIO  
- FLOATING_POINT  
- REALTIME_PRIORITY  
- PRIORITY_ENFORCE | |
|                                            | It can also be set to a combination of the above bits by using the "or" symbol (|), such as STDIO|FLOATING_POINT. Default: MASK_DEFAULT | |
| `<priority>`                              | Thread priority. The value can be specified as an unsigned integer or one of the following strings. | 0 or 1 |
|                                            | - THREAD_PRIORITY_DEFAULT  
- THREAD_PRIORITY_HIGH  
- THREAD_PRIORITY_ABOVE_NORMAL  
- THREAD_PRIORITY_NORMAL  
- THREAD_PRIORITY_BELOW_NORMAL  
- THREAD_PRIORITY_LOW | |
|                                            | When using an unsigned integer, the allowed range is platform-dependent. | |
| `<stack_size>`                            | Thread stack size, specified as an unsigned integer or set to the string THREAD_STACK_SIZE_DEFAULT. The allowed range is platform-dependent. | 0 or 1 |
53.1 Relationship Between Service Verbosity and Filter Level

A service’s verbosity influences the way the log messages reach Distributed Logger and their quantity. If a service (such as RTI Persistence Service, RTI Routing Service, or another service that is integrated with Distributed Logger) is configured with a low verbosity, it will not pass a lot of messages to Distributed Logger, even if the Distributed Logger filter level is set to a very verbose one (such as TRACE). On the contrary, a high verbosity will work better, because it will pass more messages to Distributed Logger; in this case the filter level will have more effect.

Note: Since Distributed Logger uses a separate thread to send log messages, there is little impact on performance with more verbose filter levels. However, there is some performance penalty in services that use a higher verbosity.