RTI Data Distribution Service

The Real-Time Publish-Subscribe Middleware

User's Manual

Version 4.5c

Trademarks

Copy and Use Restrictions

Third-Party Copyright Notices

Technical Support

Real-Time Innovations, Inc.
385 Moffett Park Drive
Sunnyvale, CA 94089
Phone: (408) 990-7444
Email: support@rti.com
Website: https://www.rti.com/support

Available Documentation

To get you up and running as quickly as possible, we have divided the *RTI® Data Distribution Service* documentation into several parts.

- ❏ Getting Started Guide (RTI_DDS_GettingStarted.pdf)—This document describes how to install *RTI Data Distribution Service*. It also lays out the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application. Developers should read this document first.
- ❏ **If you are using *RTI Data Distribution Service* on an embedded platform or with a database,** you will find additional documents that specifically address these configurations:
  - Addendum for Embedded Systems (RTI_DDS_GettingStarted_EmbeddedSystemsAddendum.pdf)
  - Addendum for Database Setup (RTI_DDS_GettingStarted_DatabaseAddendum.pdf).
- ❏ What's New (RTI_DDS_WhatsNew.pdf)—This document describes changes and enhancements in the current version of *RTI Data Distribution Service*. Those upgrading from a previous version should read this document first.
- ❏ Release Notes and Platform Notes (RTI_DDS_ReleaseNotes.pdf and RTI_DDS_PlatformNotes.pdf)—These documents provide system requirements, compatibility, and other platform-specific information about the product, including specific information required to build your applications using RTI, such as compiler flags and libraries.
- ❏ User's Manual (RTI_DDS_UsersManual.pdf)—This document describes the features of the product and how to use them. It is organized around the structure of the DDS APIs and certain common high-level tasks.

- ❑ **API Documentation** (ReadMe.html, RTI_DDS_ApiReference<Language>.pdf)—This extensively cross-referenced documentation, available both in HTML and printable PDF formats, is your in-depth reference to every operation and configuration parameter in the middleware. Even experienced *RTI Data Distribution Service* developers will often consult this information.

  The Programming How To's (available from the main page) provide example code. These are hyperlinked code snippets to the full API documentation, and **provide a good place to begin learning the APIs. Start by reviewing the Publication Example and Subscription Example**, which provide step-by step examples of how to send and receive data with *RTI Data Distribution Service*.

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

- ❑ **RTI Public Knowledge Base**—Accessible from http://www.rti.com/support. The Knowledge Base provides sample code, general information on *RTI Data Distribution Service*, performance information, troubleshooting tips, and other technical details.
- ❑ **RTI Customer Portal**—Accessible from http://www.rti.com/support. The portal provides a superset of the solutions available in the RTI Knowledge Base. Select the **Find Solution** link to see sample code, general information on *RTI Data Distribution Service*, performance information, troubleshooting tips, and other technical details. You must have a user name and password to access the portal; these are included in the letter confirming your purchase. If you do not have this letter, please contact **license@rti.com**.
- ❑ **RTI Example Performance Test**—This example application includes code and configuration files for testing and optimizing the performance of a simple *RTI Data Distribution Service* application on your system. The program will test both throughput and latency under a wide variety of middleware configurations. It also includes documentation on tuning the middleware and the underlying operating system.

  You can download the test from the RTI Knowledge Base, accessible from http://www.rti.com/support. In the Performance category, look for **Example Performance Test for RTI Data Distribution Service.**

  You can also review the data from several performance benchmarks here: http://www.rti.com/products/dds/benchmarks-cpp-linux.html.
- ❑ **Whitepapers and other articles**—These documents are available from http://www.rti.com/resources/.

Contents

Available Documentation iii

Welcome to RTI Data Distribution Service xxi

Conventions xxi
Extensions to the DDS Standard xxi
Environment Variables xxii
Names of Supported Platforms xxii
Additional Resources xxii

Part 1: Introduction

1 Overview 1-1

1.1 What is RTI Data Distribution Service? 1-1
1.2 What is Middleware? 1-2
1.3 Network Communications Models 1-3
1.4 Features of RTI Data Distribution Service 1-5

2 Data-Centric Publish-Subscribe Communications 2-1

2.1 What is DCPS? 2-1
2.1.1 DCPS for Real-Time Requirements 2-2
2.2 Data Types, Topics, Keys, Instances, and Samples 2-4
2.2.1 Data Topics — What is the Data Called? 2-4
2.2.2 Samples, Instances, and Keys 2-5
2.3 DataWriters/Publishers and DataReaders/Subscribers 2-7
2.4 Domains and DomainParticipants 2-10
2.5 Quality of Service (QoS) 2-11
2.5.1 Controlling Behavior with Quality of Service (QoS) Policies 2-11
2.6 Application Discovery 2-13

Part 2: Core Concepts

3 Data Types and Data Samples 3-1

3.1 Introduction to the Type System 3-4
- 3.1.1 Sequences 3-5
- 3.1.2 Strings and Wide Strings 3-6
- 3.1.3 Introduction to TypeCode 3-6

3.2 Built-in Data Types 3-7
- 3.2.1 Registering Built-in Types 3-8
- 3.2.2 Creating Topics for Built-in Types 3-8
- 3.2.3 Creating ContentFilteredTopics for Built-in Types 3-10
- 3.2.4 String Built-in Type 3-13
- 3.2.5 KeyedString Built-in Type 3-19
- 3.2.6 Octets Built-in Type 3-27
- 3.2.7 KeyedOctets Built-in Type 3-36
- 3.2.8 Managing Memory for Built-in Types 3-45
- 3.2.9 Type Codes for Built-in Types 3-51

3.3 Creating User Data Types with IDL 3-52
- 3.3.1 Variable-Length Types 3-53
- 3.3.2 Value Types 3-55
- 3.3.3 TypeCode and rtiddsgen 3-56
- 3.3.4 rtiddsgen Translations for IDL Types 3-56
- 3.3.5 Escaped Identifiers 3-75
- 3.3.6 Referring to Other IDL Files 3-76
- 3.3.7 Preprocessor Directives 3-76
- 3.3.8 Using Custom Directives 3-76

3.4 Creating User Data Types with Extensible Markup Language (XML) 3-83

3.5 Creating User Data Types with XML Schemas (XSD) 3-91
- 3.5.1 Primitive Types 3-107

3.6 Using rtiddsgen 3-107
- 3.6.1 rtiddsgen Command-Line Arguments 3-111

3.7 Using Generated Types without RTI Data Distribution Service (Standalone) 3-116
- 3.7.1 Using Standalone Types in C 3-117
- 3.7.2 Using Standalone Types in C++ 3-117
- 3.7.3 Standalone Types in Java 3-118

3.8 Interacting Dynamically with User Data Types....3-118
3.8.1 Introduction to TypeCode 3-118
3.8.2 Defining New Types....3-120
3.8.3 Sending Only a Few Fields....3-122
3.8.4 Type Extension and Versioning 3-123
3.8.5 Sending TypeCodes on the Network 3-124
3.9 Working with Data Samples....3-125
3.9.1 Objects of Concrete Types....3-125
3.9.2 Objects of Dynamically Defined Types 3-128

4 DDS Entities 4-1

4.1 Common Operations for All DDS Entities 4-2
4.1.1 Creating and Deleting Entities....4-2
4.1.2 Enabling Entities....4-3
4.1.3 Getting an Entity's Instance Handle 4-6
4.1.4 Getting Status and Status Changes 4-6
4.1.5 Getting and Setting Listeners....4-7
4.1.6 Getting the StatusCondition....4-7
4.1.7 Getting and Setting QosPolicies 4-8
4.2 QosPolicies 4-12
4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property 4-16
4.2.2 Special QosPolicy Handling Considerations for C 4-17
4.3 Statuses 4-18
4.3.1 Types of Communication Status....4-19
4.3.2 Special Status-Handling Considerations for C....4-26
4.4 Listeners....4-27
4.4.1 Types of Listeners 4-27
4.4.2 Creating and Deleting Listeners....4-29
4.4.3 Special Considerations for Listeners in C 4-30
4.4.4 Hierarchical Processing of Listeners....4-30
4.4.5 Operations Allowed within Listener Callbacks 4-32
4.5 Exclusive Areas (EAs)....4-32
4.5.1 Restricted Operations in Listener Callbacks....4-36
4.6 Conditions and WaitSets 4-37
4.6.1 Creating and Deleting WaitSets....4-38
4.6.2 WaitSet Operations 4-41

4.6.3 Waiting for Conditions 4-41
4.6.4 Processing Triggered Conditions—What to do when Wait() Returns 4-42
4.6.5 Conditions and WaitSet Example 4-44
4.6.6 GuardConditions 4-46
4.6.7 ReadConditions and QueryConditions 4-46
4.6.8 StatusConditions 4-49
4.6.9 Using Both Listeners and WaitSets 4-50

5 Topics 5-1

5.1 Topics 5-2
5.1.1 Creating Topics 5-4
5.1.2 Deleting Topics 5-5
5.1.3 Setting Topic QosPolicies 5-6
5.1.4 Copying QoS From a Topic to a DataWriter or DataReader 5-10
5.1.5 Setting Up TopicListeners 5-10
5.1.6 Navigating Relationships Among Entities 5-11

5.2 Topic QosPolicies 5-12
5.2.1 TOPIC_DATA QosPolicy 5-12

5.3 Status Indicator for Topics 5-14
5.3.1 INCONSISTENT_TOPIC Status 5-14

5.4 ContentFilteredTopics 5-15
5.4.1 Overview 5-15
5.4.2 Where Filtering is Applied—Publishing vs. Subscribing Side 5-16
5.4.3 Creating ContentFilteredTopics 5-18
5.4.4 Deleting ContentFilteredTopics 5-20
5.4.5 Using a ContentFilteredTopic 5-20
5.4.6 SQL Filter Expression Notation 5-23
5.4.7 STRINGMATCH Filter Expression Notation 5-32
5.4.8 Custom Content Filters 5-34

6 Sending Data 6-1

6.1 Preview: Steps to Sending Data 6-1

6.2 Publishers 6-3
6.2.1 Creating Publishers Explicitly vs. Implicitly 6-6
6.2.2 Creating Publishers 6-7
6.2.3 Deleting Publishers 6-9

6.2.4 Setting Publisher QosPolicies 6-9
6.2.5 Setting Up PublisherListeners 6-16
6.2.6 Finding a Publisher's Related Entities....................6-18
6.2.7 Waiting for Acknowledgments....................6-18
6.2.8 Statuses for Publishers....................6-20
6.2.9 Suspending and Resuming Publications....................6-20

6.3 DataWriters 6-20
6.3.1 Creating DataWriters....................6-24
6.3.2 Deleting DataWriters....................6-26
6.3.3 Setting Up DataWriterListeners....................6-26
6.3.4 Checking DataWriter Status....................6-28
6.3.5 Statuses for DataWriters 6-29
6.3.6 Using a Type-Specific DataWriter (FooDataWriter) 6-40
6.3.7 Writing Data 6-41
6.3.8 Flushing Batches of Data Samples 6-43
6.3.9 Writing Coherent Sets of Data Samples....................6-44
6.3.10 Waiting for Acknowledgments....................6-44
6.3.11 Managing Data Instances (Working with Keyed Data Types) 6-45
6.3.12 Setting DataWriter QosPolicies....................6-49
6.3.13 Navigating Relationships Among Entities 6-59
6.3.14 Asserting Liveliness 6-60

6.4 Publisher/Subscriber QosPolicies 6-60
6.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension)....................6-61
6.4.2 ENTITYFACTORY QosPolicy....................6-63
6.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension) 6-66
6.4.4 GROUP_DATA QosPolicy....................6-69
6.4.5 PARTITION QosPolicy 6-72
6.4.6 PRESENTATION QosPolicy....................6-79

6.5 DataWriter QosPolicies 6-85
6.5.1 BATCH QosPolicy (DDS Extension)....................6-86
6.5.2 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)....................6-92
6.5.3 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) 6-104
6.5.4 DEADLINE QosPolicy....................6-107
6.5.5 DESTINATION_ORDER QosPolicy 6-110
6.5.6 DURABILITY QosPolicy 6-113
6.5.7 DURABILITY SERVICE QosPolicy....................6-116
6.5.8 HISTORY QosPolicy....................6-118

6.5.9 LATENCYBUDGET QoS Policy 6-123
6.5.10 LIFESPAN QoS Policy 6-124
6.5.11 LIVELINESS QosPolicy 6-125
6.5.12 MULTI_CHANNEL QosPolicy (DDS Extension) 6-129
6.5.13 OWNERSHIP QosPolicy 6-132
6.5.14 OWNERSHIP_STRENGTH QosPolicy 6-136
6.5.15 PROPERTY QosPolicy (DDS Extension) 6-137
6.5.16 PUBLISH_MODE QosPolicy (DDS Extension) 6-140
6.5.17 RELIABILITY QosPolicy 6-142
6.5.18 RESOURCE_LIMITS QosPolicy 6-146
6.5.19 TRANSPORT_PRIORITY QosPolicy 6-151
6.5.20 TRANSPORT_SELECTION QosPolicy (DDS Extension) 6-152
6.5.21 TRANSPORT_UNICAST QosPolicy (DDS Extension) 6-154
6.5.22 TYPESUPPORT QosPolicy (DDS Extension) 6-157
6.5.23 USER_DATA QosPolicy 6-158
6.5.24 WRITER_DATA_LIFECYCLE QoS Policy 6-161

6.6 Flow Controllers (DDS Extension) 6-163
6.6.1 Flow Controller Scheduling Policies 6-164
6.6.2 Managing Fast DataWriters When Using a FlowController 6-166
6.6.3 Token Bucket Properties 6-166
6.6.4 Creating and Deleting FlowControllers 6-169
6.6.5 Getting and Setting Default FlowController Properties for DomainParticipants 6-170
6.6.6 Getting and Setting Properties for a Specific FlowController 6-170
6.6.7 Adding an External Trigger 6-171
6.6.8 Other FlowController Operations 6-171

7 Receiving Data 7-1

7.1 Preview: Steps to Receiving Data 7-2
7.2 Subscribers 7-4
7.2.1 Creating Subscribers Explicitly vs. Implicitly 7-9
7.2.2 Creating Subscribers 7-10
7.2.3 Deleting Subscribers 7-11
7.2.4 Setting Subscriber QosPolicies 7-12
7.2.5 Setting Up SubscriberListeners 7-19
7.2.6 Getting DataReaders with Specific Samples 7-22
7.2.7 Finding a Subscriber’s Related Entities 7-23

7.2.8 Statuses for Subscribers ... 7-24
7.3 DataReaders ... 7-25
7.3.1 Creating DataReaders ... 7-29
7.3.2 Deleting DataReaders ... 7-31
7.3.3 Setting Up DataReaderListeners ... 7-31
7.3.4 Checking DataReader Status and StatusConditions ... 7-34
7.3.5 Waiting for Historical Data ... 7-35
7.3.6 Statuses for DataReaders ... 7-35
7.3.7 Setting DataReader QosPolicies ... 7-47
7.3.8 Navigating Relationships Among Entities ... 7-53
7.4 Using DataReaders to Access Data (Read & Take) ... 7-55
7.4.1 Using a Type-Specific DataReader (FooDataReader) ... 7-55
7.4.2 Loaning and Returning Data and SampleInfo Sequences ... 7-55
7.4.3 Accessing Data Samples with Read or Take ... 7-57
7.4.4 The Sequence Data Structure ... 7-65
7.4.5 The SampleInfo Structure ... 7-66
7.5 Subscriber QosPolicies ... 7-72
7.6 DataReader QosPolicies ... 7-72
7.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension) ... 7-72
7.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) ... 7-78
7.6.3 READER_DATA_LIFECYCLE QoS Policy ... 7-84
7.6.4 TIME_BASED_FILTER QosPolicy ... 7-87
7.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension) ... 7-90

8 Working with Domains ... 8-1

8.1 Fundamentals of Domains and DomainParticipants ... 8-1
8.2 DomainParticipantFactory ... 8-4
8.2.1 Setting DomainParticipantFactory QosPolicies ... 8-7
8.2.2 Getting and Setting Default QoS for DomainParticipants ... 8-8
8.2.3 Freeing Resources Used by the DomainParticipantFactory ... 8-9
8.2.4 Looking Up a DomainParticipant ... 8-9
8.2.5 Getting QoS Values from a QoS Profile ... 8-9
8.3 DomainParticipants ... 8-10
8.3.1 Creating a DomainParticipant ... 8-16
8.3.2 Deleting DomainParticipants ... 8-18
8.3.3 Deleting Contained Entities ... 8-18

8.3.4 Choosing a Domain ID and Creating Multiple Domains 8-19
8.3.5 Setting Up DomainParticipantListeners 8-20
8.3.6 Setting DomainParticipant QosPolicies 8-22
8.3.7 Looking up Topic Descriptions 8-29
8.3.8 Finding a Topic 8-30
8.3.9 Getting the Implicit Publisher or Subscriber 8-30
8.3.10 Asserting Liveliness 8-32
8.3.11 Learning about Discovered DomainParticipants 8-32
8.3.12 Learning about Discovered Topics 8-33
8.3.13 Other DomainParticipant Operations 8-33

8.4 DomainParticipantFactory QosPolicies 8-34
8.4.1 PROFILE QosPolicy (DDS Extension) 8-34
8.4.2 SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension) 8-36

8.5 DomainParticipant QosPolicies 8-38
8.5.1 DATABASE QosPolicy (DDS Extension) 8-38
8.5.2 DISCOVERY QosPolicy (DDS Extension) 8-42
8.5.3 DISCOVERY_CONFIG QosPolicy (DDS Extension) 8-47
8.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) 8-54
8.5.5 ENTITYNAME QosPolicy (DDS Extension) 8-60
8.5.6 EVENT QosPolicy (DDS Extension) 8-61
8.5.7 RECEIVER_POOL QosPolicy (DDS Extension) 8-64
8.5.8 TRANSPORT_BUILTIN QosPolicy (DDS Extension) 8-66
8.5.9 WIRE_PROTOCOL QosPolicy (DDS Extension) 8-67

8.6 Clock Selection 8-75
8.6.1 Available Clocks 8-75
8.6.2 Clock Selection Strategy 8-76

9 Building Applications 9-1

9.1 Running on a Computer Not Connected to a Network 9-2

9.2 RTI Data Distribution Service Header Files — All Architectures 9-2

9.3 UNIX-based Platforms 9-3
9.3.1 Required Libraries 9-4
9.3.2 Compiler Flags 9-4

9.4 Windows Platforms 9-5
9.4.1 Using Microsoft Visual C++ 6.0 9-5

9.4.2 Using Visual Studio .NET, Visual Studio .NET 2003, or Visual Studio 2005......9-7
9.5 Java Platforms......9-8
9.5.1 Java Libraries......9-8
9.5.2 Native Libraries......9-8

Part 3: Advanced Concepts

10 Reliable Communications......10-1

10.1 Sending Data Reliably......10-1
10.1.1 Best-effort Delivery Model......10-2
10.1.2 Reliable Delivery Model......10-2
10.2 Overview of the Reliable Protocol......10-4
10.3 Using QosPolicies to Tune the Reliable Protocol......10-9
10.3.1 Enabling Reliability......10-10
10.3.2 Tuning Queue Sizes and Other Resource Limits......10-11
10.3.3 Controlling Queue Depth with the History QosPolicy......10-18
10.3.4 Controlling Heartbeats and Retries with the DataWriterProtocol QosPolicy......10-18
10.3.5 Avoiding Message Storms with DataReaderProtocol QosPolicy......10-27
10.3.6 Resending Samples to Late-Joiners with the Durability QosPolicy......10-27
10.3.7 Use Cases......10-28

11 Mechanisms for Achieving Information Durability and Persistence11-1

11.1 Introduction......11-1
11.1.1 Scenario 1. A DataReader Joins after a DataWriter Restarts (Durable Writer History)......11-3
11.1.2 Scenario 2: A DataReader Restarts While DataWriter Stays Up (Durable Reader State)......11-4
11.1.3 Scenario 3. A DataReader Joins after the DataWriter Leaves the Domain (Durable Data)......11-4
11.2 Durability and Persistence Based on Virtual GUIDs......11-6
11.3 Durable Writer History......11-7
11.3.1 Durable Writer History Use Case......11-8

11.3.2 How To Configure Durable Writer History 11-9
11.4 Durable Reader State 11-13
11.4.1 Durable Reader State Use Case 11-15
11.4.2 How To Configure a DataReader for Durable Reader State 11-16
11.5 Data Durability 11-18
11.5.1 RTI Persistence Service 11-18

12 Discovery 12-1

12.1 What is Discovery? 12-2
12.1.1 Simple Participant Discovery 12-2
12.1.2 Simple Endpoint Discovery 12-3
12.2 Configuring the Peers List Used in Discovery 12-4
12.2.1 Peer Descriptor Format 12-6
12.2.2 NDDS_DISCOVERY_PEERS Environment Variable Format 12-8
12.2.3 NDDS_DISCOVERY_PEERS File Format 12-9
12.3 Discovery Implementation 12-10
12.3.1 Participant Discovery 12-11
12.3.2 Endpoint Discovery 12-20
12.3.3 Discovery Traffic Summary 12-26
12.3.4 Discovery-Related QoS 12-27
12.4 Debugging Discovery 12-28
12.5 Ports Used for Discovery 12-31
12.5.1 Inbound Ports for Meta-Traffic 12-32
12.5.2 Inbound Ports for User Traffic 12-33
12.5.3 Automatic Selection of participant_id and Port Reservation 12-33
12.5.4 Tuning domain_id_gain and participant_id_gain 12-34

13 Transport Plugins 13-1

13.1 Builtin Transport Plugins 13-2
13.2 Extension Transport Plugins 13-3
13.3 The NDDSTransportSupport Class 13-4
13.4 Explicitly Creating Builtin Transport Plugin Instances 13-4
13.5 Setting Builtin Transport Properties of the Default Transport Instance—get/set_builtin_transport_properties() 13-5
13.6 Setting Builtin Transport Properties with the PropertyQosPolicy 13-7

13.6.1 Notes Regarding Loopback and Shared Memory 13-25
13.6.2 Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6 13-25

13.7 Installing Additional Builtin Transport Plugins with register_transport() 13-26
13.7.1 Transport Lifecycles 13-27
13.7.2 Transport Aliases 13-28
13.7.3 Transport Network Addresses 13-29

13.8 Installing Additional Builtin Transport Plugins with PropertyQosPolicy 13-29

13.9 Other Transport Support Operations 13-31
13.9.1 Adding a Send Route 13-31
13.9.2 Adding a Receive Route 13-32
13.9.3 Looking Up a Transport Plugin 13-33

14 Built-In Topics 14-1

14.1 Listeners for Built-in Entities 14-2

14.2 Built-in DataReaders 14-2
14.2.1 LOCATOR_FILTER QoS Policy (DDS Extension) 14-11

14.3 Accessing the Built-in Subscriber 14-12

14.4 Restricting Communication—Ignoring Entities 14-13
14.4.1 Ignoring Specific Remote DomainParticipants 14-14
14.4.2 Ignoring Publications and Subscriptions 14-14
14.4.3 Ignoring Topics 14-16

15 Configuring QoS with XML 15-1

15.1 Example XML File 15-2

15.2 How to Load XML-Specified QoS Settings 15-3
15.2.1 Loading, Reloading and Unloading Profiles 15-4

15.3 How to Use XML-Specified QoS Settings 15-5

15.4 XML File Syntax 15-8

15.5 XML String Syntax 15-9

15.6 How the XML is Validated 15-10
15.6.1 Validation at Run-Time 15-10
15.6.2 XML File Validation During Editing 15-11

15.7 Configuring QoS with XML 15-12
15.7.1 QosPolicies 15-13

15.7.2 Sequences 15-13
15.7.3 Arrays 15-16
15.7.4 Enumeration Values 15-16
15.7.5 Time Values (Durations) 15-17
15.7.6 Transport Properties 15-17
15.7.7 Thread Settings 15-18
15.8 QoS Profiles 15-18
15.8.1 QoS Profiles with a Single QoS 15-20
15.8.2 QoS-Profile Inheritance 15-20
15.8.3 Topic Filters 15-23
15.8.4 Overwriting Default QoS Values 15-27
15.8.5 Get Qos Profiles 15-28
15.9 QoS Libraries 15-28
15.9.1 Get Qos Profile Libraries 15-30
15.10URL Groups 15-30

16 Multi-channel DataWriters 16-1
16.1 What is a Multi-channel DataWriter? 16-3
16.2 How to Configure a Multi-channel DataWriter 16-5
16.2.1 Limitations 16-5
16.3 Multi-channel Configuration on the Reader Side 16-7
16.4 Where Does the Filtering Occur? 16-9
16.4.1 Filtering at the DataWriter 16-9
16.4.2 Filtering at the DataReader 16-10
16.4.3 Filtering on the Network Hardware 16-10
16.5 Fault Tolerance and Redundancy 16-11
16.6 Reliability with Multi-Channel DataWriters 16-12
16.6.1 Reliable Delivery 16-12
16.6.2 Reliable Protocol Considerations 16-12
16.7 Performance Considerations 16-13
16.7.1 Network-Switch Filtering 16-13
16.7.2 DataWriter and DataReader Filtering 16-14

17 RTI Data Distribution Service Threading Model 17-1
17.1 Database Thread 17-2

17.2 Event Thread....17-3
17.3 Receive Threads....17-4
17.4 Exclusive Areas, RTI Data Distribution Service Threads and User Listeners....17-6
17.5 Controlling CPU Core Affinity for RTI Threads....17-7

18 Troubleshooting....18-1

18.1 What Version am I Running?....18-1
18.1.1 Finding Version Information in Revision Files....18-1
18.1.2 Finding Version Information Programmatically....18-2
18.2 Controlling Messages from RTI Data Distribution Service....18-3
18.2.1 Format of Logged Messages....18-6

Part 4: RTI Secure WAN Transport

19 RTI Secure WAN Transport....19-1

19.1 WAN Traversal via UDP Hole-punching....19-3
19.1.1 Protocol Details....19-4
19.2 WAN Locators....19-9
19.3 Datagram Transport-Layer Security (DTLS)....19-10
19.3.1 Security Model....19-10
19.3.2 Liveliness Mechanism....19-11
19.4 Certificate Support....19-11
19.5 License Issues....19-12

20 Configuring RTI Secure WAN Transport....20-1

20.1 Example Applications....20-1
20.2 Setting Up a Transport with the Property QoS....20-2
20.3 WAN Transport Properties....20-4
20.4 Secure Transport Properties....20-13
20.5 Explicitly Instantiating a WAN or Secure Transport Plugin....20-17
20.5.1 Additional Header Files and Include Directories....20-18
20.5.2 Additional Libraries....20-18

20.5.3 Compiler Flags 20-18

Part 5: RTI Persistence Service

21 Introduction to RTI Persistence Service 21-1

22 Configuring RTI Persistence Service 22-1

22.1 How to Load the XML Configuration 22-2
22.2 XML Configuration File 22-3
22.2.1 Configuration File Syntax 22-4
22.2.2 XML Validation 22-5
22.3 QoS Configuration 22-7
22.4 Configuring the RTI Persistence Service Application 22-8
22.5 Configuring the Persistent Storage 22-10
22.6 Configuring Participants 22-12
22.7 Creating Persistence Groups 22-13
22.7.1 QoSs 22-17
22.7.2 Durability Service Qos Policy 22-17
22.7.3 Sharing a Publisher/Subscriber 22-18
22.7.4 Sharing a Database Connection 22-18

23 Running RTI Persistence Service 23-1

23.1 Starting RTI Persistence Service 23-1
23.2 Stopping RTI Persistence Service 23-2

Part 6: RTI CORBA Compatibility Kit

24 Introduction to RTI CORBA Compatibility Kit 24-1

25 Generating CORBA-Compatible Code with rtiddsgen 25-1

25.1 Generating C++ Code 25-2
25.2 Generating Java Code 25-3

26 Supported IDL Types 26-1

Part 7: RTI RTSJ Extension Kit

27 Introduction to RTI RTSJ Extension Kit 27-1

28 Using RTI RTSJ Extension Kit 28-1

Part 8: RTI TCP Transport

29 Configuring the RTI TCP Transport 29-1

29.1 TCP Communication Scenarios 29-1
29.1.1 Communication Within a Single LAN 29-1
29.1.2 Symmetric Communication Across NATs 29-2
29.1.3 Asymmetric Communication Across NATs 29-4
29.2 Configuring the RTI TCP Transport 29-6
29.2.1 Choosing a Transport Mode 29-6
29.2.2 Explicitly Instantiating the TCP Transport Plugin 29-7
29.2.3 Configuring the RTI TCP Transport with the Property QosPolicy 29-9
29.2.4 Setting the Initial Peers 29-11
29.2.5 TCP/TLS Transport Properties 29-11

Welcome to RTI Data Distribution Service

Welcome to *RTI Data Distribution Service*—the real-time network middleware from Real-Time Innovations, Inc (RTI). *RTI Data Distribution Service* provides an easy way to build applications that communicate over a network. With *RTI Data Distribution Service*, applications can transfer information between different architectures and operating systems without any changes to the application code. *RTI Data Distribution Service* is designed for real-time applications that need control over timing and memory usage, have low latency requirements, or need high robustness to network changes or failures. *RTI Data Distribution Service* consists of a library that is linked to your applications.

Conventions

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

C, C++/CLI, C#, and Java APIs are also available; they are fully described in the online (HTML) documentation.

Namespace support in C++, C++/CLI, and C# is also available; see the online documentation (from the **Modules** page, select **Using DDS:: Namespace**) for details.

Extensions to the DDS Standard

RTI Data Distribution Service 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 *RTI Data Distribution Service*, such as **NDDSTransportSupport::register_transport()**.

Environment Variables

RTI Data Distribution Service documentation refers to pathnames that have been customized during installation. **NDDSHOME** refers to the installation directory of *RTI Data Distribution Service*.

Names of Supported Platforms

RTI Data Distribution Service runs on several different target platforms. To support this vast array of platforms, *RTI Data Distribution Service* 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, **i86Linux2.4gcc3.2** is the directory that contains files specific to Linux® version 2.4 for the Intel processor, compiled with gcc version 3.2.

For a full list of supported platforms, see the Platform Notes.

Additional Resources

The details of each API (such as function parameters, return values, etc.) and examples are in the online documentation. In case of discrepancies between the information in this document and the online documentation, the latter should be considered more up-to-date.

Part 1: Introduction

This introduces the general concepts behind data-centric publish-subscribe communications and provides a brief tour of *RTI Data Distribution Service*.

- Chapter 1: Overview
- Chapter 2: Data-Centric Publish-Subscribe Communications

Chapter 1 Overview

RTI Data Distribution Service is network middleware for distributed real-time applications. *RTI Data Distribution Service* simplifies application development, deployment and maintenance and provides fast, predictable distribution of time-critical data over a variety of transport networks.

With *RTI Data Distribution Service* middleware, you can:

- ❏ Perform complex one-to-many and many-to-many network communications; the application uses an OMG standard API, *DDS*, to publish and subscribe to the data.
- ❏ 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 chapter introduces basic concepts of middleware and common communication models, and describes how *RTI Data Distribution Service*'s feature-set addresses the needs of real-time systems.

1.1 What is RTI Data Distribution Service?

RTI Data Distribution Service is network middleware 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. *RTI Data Distribution Service* uses the publish-subscribe communications model to make data distribution efficient and robust.

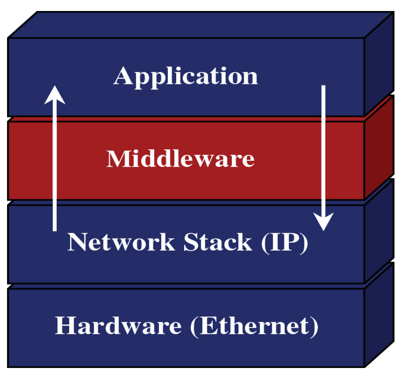
RTI Data Distribution Service implements the Data-Centric Publish-Subscribe (DCPS) API within the OMG's Data Distribution Service (DDS) for Real-Time Systems. DDS is the first standard developed for the needs of real-time systems. DCPS provides an efficient way to transfer data in a distributed system.

With *RTI Data Distribution Service*, 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. *RTI Data Distribution Service* is highly configurable so programmers can adapt it to meet the application's specific communication requirements.

1.2 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 1.1). 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 1.1 **Network Middleware**



RTI Data Distribution Service is middleware that insulates applications from the raw operating-system network stack.

Publish-subscribe middleware *RTI Data Distribution Service* is based on a publish-subscribe communications model. Publish-subscribe (PS) middleware 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 of all those special cases. What experienced network middleware developers know is that handling special cases accounts for over 80% of the effort and code.

1.3 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.

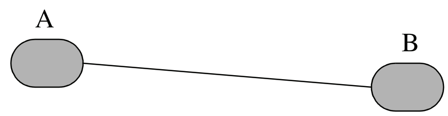
This section describes three main types of network communications models:

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

Point-to-point model Point-to-point is the simplest form of communication, as illustrated in Figure 1.2. 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.

Figure 1.2 **Point-to-Point**

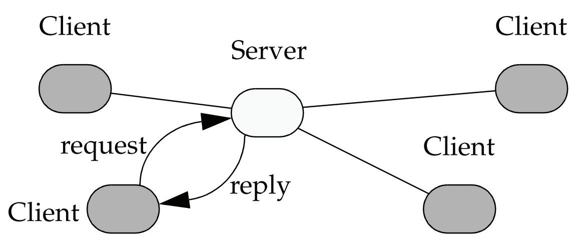


Point-to-point is one-to-one communications.

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 1.3. 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 (request), the parlor asks the client where the response (pizza) should be sent. 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.

Figure 1.3 **Client-Server**



Client-server is many-to-one communications.

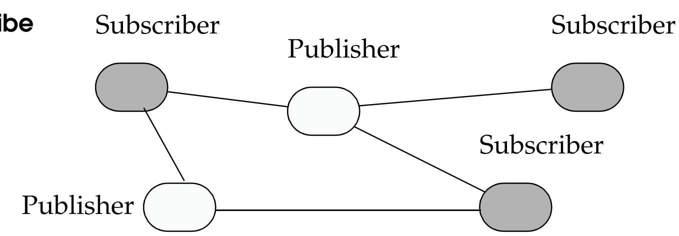
Publish-subscribe model In the publish-subscribe communications model, computer applications (nodes) “subscribe” to data they need and “publish” data they want to share. Messages pass directly between the publisher and the subscribers, rather than moving into and out of a centralized server. Most time-sensitive 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)

Figure 1.4 **Publish-Subscribe**



Publish-subscribe is many-to-many communications.

1.4 Features of RTI Data Distribution Service

RTI Data Distribution Service supports mechanisms that go beyond the basic publish-subscribe model. The key benefit is that applications that use *RTI Data Distribution Service* 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. *RTI Data Distribution Service* automatically handles all aspects of message delivery, without requiring any intervention from the user applications, including:

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

This is made possible by how *RTI Data Distribution Service* 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, *RTI Data Distribution Service* greatly simplifies distributed application design and encourages modular, well-structured programs.

Furthermore, *RTI Data Distribution Service* 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 (*domains*) each using *RTI Data Distribution Service* can be used over the same physical network. Applications are only able to participate in the domains to which they belong. Individual applications can be configured to participate in multiple 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. *RTI Data Distribution Service* 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.
- **Multiple Built-in Transports** Includes UDP/IP and shared memory transports.
- **Multi-language support** Includes APIs for the C, C++, C++/CLI, C#, and Java™ programming languages.
- **Multi-platform support** Includes support for flavors of UNIX® (Linux® and Solaris™), real-time operating systems (INTEGRITY®, VxWorks®, QNX®, and LynxOS®), and Windows® (2000, 2003, CE, Vista, and XP). (Consult the 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 chapter describes the formal communications model used by *RTI Data Distribution Service*, 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 Section 1.3.

This chapter includes the following sections:

- What is DCPS? (Section 2.1)
- Data Types, Topics, Keys, Instances, and Samples (Section 2.2)
- DataWriters/Publishers and DataReaders/Subscribers (Section 2.3)
- Domains and DomainParticipants (Section 2.4)
- Quality of Service (QoS) (Section 2.5)
- Application Discovery (Section 2.6)

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. *RTI Data Distribution Service* offers C, 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 *RTI Data Distribution Service* 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 publication data 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 *RTI Data Distribution Service*, 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 data 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 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 data 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.

RTI Data Distribution Service provides user-level configuration of its internal threads that process incoming data. Users may configure *RTI Data Distribution Service* 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 *RTI Data Distribution Service*, was designed and implemented specifically to address the requirements above through configuration parameters known as QosPolicies defined by the DCPS standard (see QosPolicies (Section 4.2)). The following section introduces basic DCPS terminology and concepts.

2.2 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:

```
struct Time {
    long year;
    short day;
    short hour;
    short minute;
    short second;
};

struct StockPrice {
    float price;
    Time timeStamp;
};
```

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 Chapter 3.

2.2.1 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 *Topic*s may refer to the same data type.

*Topic*s interconnect *DataWriters* and *DataReaders*. A *DataWriter* is an object in an application that tells *RTI Data Distribution Service* (and indirectly, other applications) that it has some values of a certain *Topic*. A corresponding *DataReader* is an object in an application that tells *RTI Data Distribution Service* 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 *DataReaders* are described more in Section 2.3.

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 *Topic*s 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

```
struct StockPrice {
    float  price;
    Time    timeStamp;
};
```

Topic: "IBM"

Topic: "MSFT"

Topic: "GE"

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.

2.2.2 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 *samples***.** In our stock-price example, samples show the price of a stock at a certain point in time. So each 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, 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. S*amples* are then updates to particular instances of a topic. Applications can subscribe to a topic and receive samples for many different instances. Applications can publish 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, 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 samples of an instance of "StockPrices", with each instance corresponding to a single stock symbol.

Data Type: StockPrice

```
struct StockPrice {
    float  price;
    Time   timeStamp;
    char   *symbol;   //@key
};
```

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

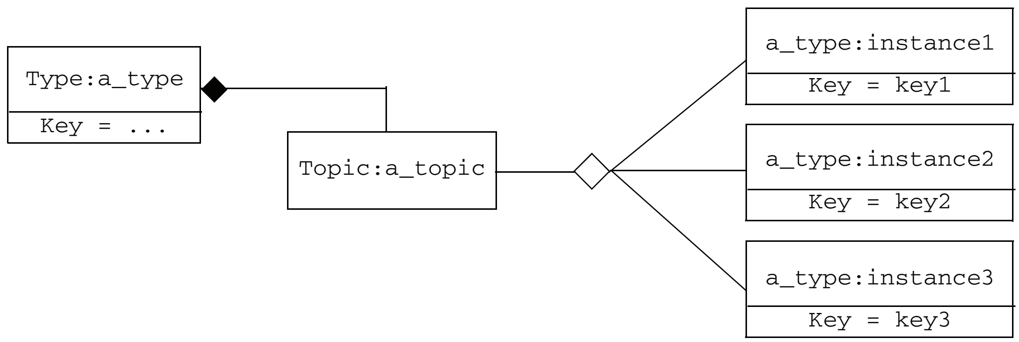
sample b, price = $73.50

Etc.

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.

To summarize, the unique values of data being passed using DCPS are called *samples*. A *sample* is a combination of a *Topic* (distinguished by a *Topic* name), an *instance* (distinguished by a *key*), and the actual *user data* of a certain data type. As seen in Figure 2.1 on page 2-7, 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 Chapter 3: Data Types and Data Samples and Chapter 5: Topics.

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



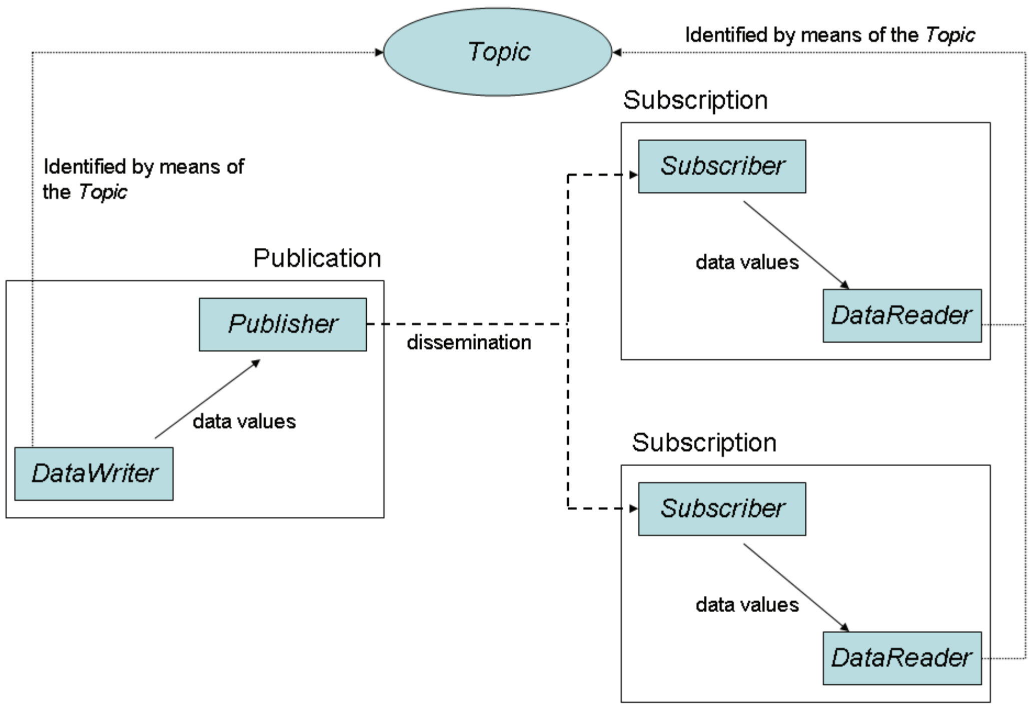
By using keys, a Topic can identify a collection of data-object instances.

2.3 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*, *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 Chapter 4: DDS Entities.

The sending side uses objects called *Publishers* and *DataWriters*. The receiving side uses objects called *Subscribers* and *DataReaders*. Figure 2.2 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 different data types. When user

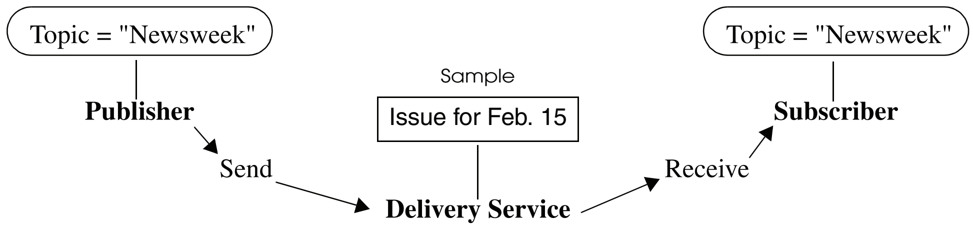
code calls the **write()** method on a *DataWriter*, the data sample is passed to the *Publisher* object which does the actual dissemination of data on the network. For more information, see Chapter 6: Sending Data.

- 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 *DataReaders* to access data received over DCPS. A *DataReader* is associated with a single *Topic*. You can have multiple *DataReaders* 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 *DataReaders*. A *DataReader* can only be owned by a single *Subscriber* while a *Subscriber* can own many *DataReaders*. 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 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 Chapter 7: Receiving Data.
- 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 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. Note that by subscribing to a publication, subscribers are requesting current and future samples of that publication (such as once a week in the case of *Newsweek*), so that as new samples are published, they are delivered without having to submit another request for data.

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

Figure 2.3 **An Example of Publish-Subscribe**



The publish-subscribe model is analogous to publishing magazines. The Publisher sends samples of a particular Topic to all Subscribers of that Topic. With Newsweek® *magazine, the Topic would be "Newsweek." The sample consists of the data (articles and pictures) sent to all Subscribers every week. The middleware (RTI Data Distribution Service) is the distribution channel: all of the planes, trucks, and people who distribute the weekly issues to the Subscribers.*

2.4 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 *Domains*.

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

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

An application may belong to multiple domains simultaneously by creating multiple *DomainParticipants* with different domain indices. However, *Publishers*/*DataWriters* and *Subscribers*/*DataReaders* only belong to the domain in which they were created.

As mentioned before, multiple domains may be used for application isolation which is useful when users are testing their 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, domains may be a way to scale and construct larger systems that are composed of multi-node subsystems. Each subsystem would use an internal domain for intra-system communications and an external domain to connect to other subsystems.

For more information, see Chapter 8: Working with Domains.

2.5 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.

2.5.1 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 *Topic*s, *DataWriters*, *Publishers*, *DataReaders*, *Subscriber*s, 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 data 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 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 *DataWriter*s 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 QosPolicies (Section 4.2).

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*, *RTI Data Distribution Service* 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 incompatible. 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 *RTI Data Distribution Service* 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* can not satisfy the requested QosPolicies of a *DataReader*, *RTI Data Distribution Service* will not connect the two entities and will notify the applications on each side of the incompatibility if so configured.

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, *RTI Data Distribution Service* 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*. *RTI Data Distribution Service* 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 *RTI Data Distribution Service*, see QosPolicies (Section 4.2).

2.6 Application Discovery

The DCPS model provides anonymous, transparent, many-to-many communications. Each time an application sends a sample of a particular *Topic*, the middleware distributes the 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, *RTI Data Distribution Service* 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, *RTI Data Distribution Service* must keep a list of applications and publications for each of the *Topics* to which the application has subscribed.

This propagation of this information (the existence of publications and subscriptions and associated QoS) between applications by *RTI Data Distribution Service* is known as the *discovery* process. While the DDS (DCPS) standard does not specify how discovery occurs, *RTI Data Distribution Service* uses a standard protocol RTPS for both discovery and formatting on-the-wire packets.

When a *DomainParticipant* is created, *RTI Data Distribution Service* sends out packets on the network to announce its existence. When an application finds out that another application belongs to the same 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 Chapter 12: Discovery.

Part 2: Core Concepts

This section includes the following chapters:

- ❏ Chapter 3: Data Types and Data Samples
- ❏ Chapter 4: DDS Entities
- ❏ Chapter 5: Topics
- ❏ Chapter 6: Sending Data
- ❏ Chapter 7: Receiving Data
- ❏ Chapter 8: Working with Domains
- ❏ Chapter 9: Building Applications

Chapter 3 Data Types and Data Samples

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.3.2.2/Solaris/Sparc) and transparently deliver it to another (for example, Java/JDK 1.6/Windows XP/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.

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

To publish and/or subscribe to data with *RTI Data Distribution Service*, 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 Built-in Data Types (Section 3.2).

   - Use the RTI code generator, *rtiddsgen,* 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 code generator accepts input in a number of formats to make it easy to integrate *RTI Data Distribution Service* with your development processes and IT infrastructure:

     - **OMG IDL.** This format is a standard component of both the DDS and CORBA specifications. It describes data types with a C++-like syntax. This format is described in Creating User Data Types with IDL (Section 3.3).

     - **XML schema (XSD)**, either independent or embedded in a WSDL file. XSD should be the format of choice for those using *RTI Data Distribution Service* alongside or connected to a web-services infrastructure. This format is described in Creating User Data Types with XML Schemas (XSD) (Section 3.5).

     - **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. This format is described in Creating User Data Types with Extensible Markup Language (XML) (Section 3.4).

- 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 Defining New Types (Section 3.8.2).

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 Defining New Types (Section 3.8.2).

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 Chapter 5: Topics.

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 Chapter 6: Sending Data and Chapter 7: Receiving Data, respectively.

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

This chapter describes:

❏ Introduction to the Type System (Section 3.1 on Page 3-4)

❏ Built-in Data Types (Section 3.2 on Page 3-7)

❏ Creating User Data Types with IDL (Section 3.3 on Page 3-52)

❏ Creating User Data Types with Extensible Markup Language (XML) (Section 3.4 on Page 3-83)

❏ Creating User Data Types with XML Schemas (XSD) (Section 3.5 on Page 3-91)

❏ Using rtiddsgen (Section 3.6 on Page 3-107)

❏ Using Generated Types without RTI Data Distribution Service (Standalone) (Section 3.7 on Page 3-116)

❏ Using rtiddsgen (Section 3.6 on Page 3-107)

❏ Working with Data Samples (Section 3.9 on Page 3-125)

3.1 Introduction to the Type System

A *user data type* is any custom type that your application defines for use with *RTI Data Distribution Service*. 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 *RTI Data Distribution Service*; 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 identical.

- ❑ octet, char, wchar
- ❑ short, unsigned short
- ❑ long, unsigned long
- ❑ long long, unsigned long long
- ❑ 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

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.

- bitfield[1]
- union
- struct
- value type, a complex type that supports inheritance and other object-oriented features

To use a data type with *RTI Data Distribution Service*, you must define that type in a way the middleware understands and then register the type with the middleware. These steps allow *RTI Data Distribution Service* 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 *Sequence Support* section under the *Infrastructure Module* in the online API documentation.

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

C++ users will find sequences conceptually similar to the **deque** class in the Standard Template Library (STL).

Elements in a sequence are accessed with their index, just like elements in an array. Indices start from zero. 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 greatest value its maximum may take. The bound is very important because it allows *RTI Data Distribution Service* 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 had no bound, *RTI Data Distribution Service* would not know how large to allocate its buffers and would therefore have to allocate them on the fly as individual samples were read and written—severely impacting the latency and determinism of your application. Therefore, *RTI Data Distribution Service* supports only bounded sequences; any unbounded sequences found in an IDL file will

1. Data types containing bitfield members are not supported by DynamicData.

be given a default bound of 100 elements (see rtiddsgen Command-Line Arguments (Section 3.6.1)).

3.1.2 Strings and Wide Strings

RTI Data Distribution Service, 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 *RTI Data Distribution Service* are four bytes long, large enough to store not only two-byte Unicode/UTF16 characters but also UTF32 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++).

3.1.3 Introduction to TypeCode

Type schemas—the names and definitions of a type and its fields—are represented by TypeCode objects. 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.

```
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_LONGLONG,
  TK_ULONGLONG,
  TK_LONGDOUBLE,
  TK_WCHAR,
  TK_WSTRING,
  TK_VALUE,
  TK_SPARSE
}
```

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 online documentation (select **Modules**, **Topic Module, Type Code Support**).

3.1.3.1 Sending TypeCodes on the Network

In addition to being used locally, serialized type codes are typically published automatically during discovery as part of the built-in topics for publications and subscriptions. See Built-in DataReaders (Section 14.2). This allows applications to publish or subscribe to topics of arbitrary types. This functionality is useful for generic system monitoring tools like the *rtiddsspy* debug tool (in the online documentation, select **Modules**, **Programming Tools**).

Note: Type codes are not cached by *RTI Data Distribution Service* upon receipt and are therefore not available from the built-in data returned by the *DataWriter's* **get_matched_subscription_data()** operation or the *DataReader's* **get_matched_publication_data()** operation.

If your data type has an especially complex type code, you may need to increase the value of the **type_code_max_serialized_length** field in the *DomainParticipant's* DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4). Or, to prevent the propagation of type codes altogether, you can set this value 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 *TypeCode* propagation.

3.2 Built-in Data Types

RTI Data Distribution Service provides a set of standard types that are built into the middleware. These types can be used immediately; they do not require writing IDL, invoking the *rtiddsgen* utility (see Section 3.6), or using the dynamic type API (see Section 3.2.8).

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. User Data Types

The built-in type API is located under the DDS namespace in C++ and .Net. For Java, the API is contained inside the package **com.rti.dds.type.builtin**.

built-in data types are discussed in following sections:

- Registering Built-in Types (Section 3.2.1)
- Creating Topics for Built-in Types (Section 3.2.2)
- Creating ContentFilteredTopics for Built-in Types (Section 3.2.3)
- String Built-in Type (Section 3.2.4)
- KeyedString Built-in Type (Section 3.2.5)
- Octets Built-in Type (Section 3.2.6)
- KeyedOctets Built-in Type (Section 3.2.7)
- String Built-in Type (Section 3.2.4)

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 PROPERTY QosPolicy (DDS Extension) (Section 6.5.15).

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 Creating Topics (Section 5.1.1)); 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 C++: <*BuiltinType*> = String, KeyedString, Octets or KeyedOctets
For Java and .Net: <*BuiltinType*> = String, KeyedString, Bytes or KeyedBytes

C API:

```
const char* DDS_<BuiltinType>TypeSupport_get_type_name();
```

C++ API with namespace:

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

C++ API without namespace:

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

C++/CLI API:

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

C# API:

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

Java API:

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

3.2.2.1 Topic Creation Examples

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

C Example:

```
DDS_Topic * 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);
```

C++ Example with Namespaces:

```
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);
```

C++/CLI Example:

```
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:

```
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);
```

Java Example:

```
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 Creating ContentFilteredTopics for Built-in Types

To create a ContentFilteredTopic for a built-in type, just use the standard *DomainParticipant* operations, **create_contentfilteredtopic()** or **create_contentfilteredtopic_with_filter** (see Section 5.4.3).

The field names used in the filter expressions for the built-in SQL (see Section 5.4.6) and StringMatch filters (see Section 5.4.7) must correspond to the names provided in the IDL description of the built-in types (see Section 3.2.4).

3.2.3.1 ContentFilteredTopic Creation Examples

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

C Example:

```
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_StringTypeSupport_get_type_name(),
                         &DDS_TOPIC_QOS_DEFAULT,
                         NULL, DDS_STATUS_MASK_NONE);

contentFilteredTopic =
       DDS_DomainParticipant_create_contentfilteredtopic(
                         participant, "StringContentFilteredTopic",
                         topic, "value = 'Hello World!'",
                         &parameters);
```

C++ Example with Namespaces:

```
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);
```

3. User Data Types

C++/CLI Example:

```
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:

```
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:

```
import com.rti.dds.type.builtin.*;
...

/* Create a String ContentFilteredTopic */
Topic topic = participant.create_topic(
                 "StringTopic", StringTypeSupport.get_type_name(),
                  DomainParticipant.TOPIC_QOS_DEFAULT,
                  null, StatusKind.STATUS_MASK_NONE);

StringSeq parameters = new StringSeq();

ContentFilteredTopic contentFilteredTopic =
    participant.create_contentfilteredtopic(
                     "StringContentFilteredTopic", topic,
                     "value = 'Hello World!'", parameters);
```

3.2.4 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. This type can be used to publish and subscribe to a single string.

3.2.4.1 Creating and Deleting Strings

In C and C++, *RTI Data Distribution Service* provides a set of operations to create (**DDS::String_alloc()**), destroy (**DDS::String_free()**), and clone strings (**DDS::String_dup()**). Select **Modules, DDS API Reference, Infrastructure Module, String support** in the online documentation for more details.

Memory Considerations in Copy Operations:

When the read/take operations that take a sequence of strings as a parameter are used in copy mode, *RTI Data Distribution Service* 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 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.

3.2.4.2 String DataWriter

The string *DataWriter* API matches the standard *DataWriter* API (see Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.6)). There are no extensions.

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:

```
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);
```

C++ Example with Namespaces:

```
#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);
```

C++/CLI Example:

```
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);
```

C# Example:

```
using System;
using 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);
```

Java Example:

```
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.HANDLE_NIL);
String str = "Hello World!";
stringWriter.write(str, InstanceHandle_t.HANDLE_NIL);
```

3.2.4.3 String DataReader

The string *DataReader* API matches the standard *DataReader* API (see Using a Type-Specific DataReader (FooDataReader) (Section 7.4.1)). 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:

```
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(&data_seq); ++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);
```

C++ Example with Namespaces:

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

StringSeq dataSeq;
SampleInfoSeq infoSeq;
StringDataReader * 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) {
        StringTypeSupport::print_data(dataSeq[i]);
    }
}
/* Return loan */
retCode = stringReader->return_loan(dataSeq, infoSeq);
```

C++/CLI Example:

```
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);
```

C# Example:

```
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 < 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);
```

Java Example:

```
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 < data_seq.length(); ++i) {
    if (((SampleInfo)infoSeq.get(i)).valid_data) {
        System.out.println((String)dataSeq.get(i));
    }
}
/* Return loan */
stringReader.return_loan(dataSeq, infoSeq);
```

3.2.5 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/C++ Representation (without namespaces):

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

C++/CLI Representation:

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

C# Representation:

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

Java Representation:

```
package com.rti.dds.type.builtin;
class KeyedString {
    public String key;
    public String value;
    ...
}
```

3.2.5.1 Creating and Deleting Keyed Strings

RTI Data Distribution Service provides a set of constructors/destructors to create/destroy Keyed Strings. Select **Modules, DDS API Reference, Topic Module, Built-in Types** in online documentation for details.

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 the online documentation (select **Modules, DDS API Reference, Infrastructure Module, String Support**).

3.2.5.2 Keyed String DataWriter

The keyed string *DataWriter* API is extended with the following methods (in addition to the standard methods described in Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.6)):

```
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 operations in all supported languages, see the online documentation.

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:

```
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_string_w_key(
                                            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:

```
#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;

/* Write some data using individual strings */
retCode = stringWriter->write("Key 1", "Value 1", HANDLE_NIL);

char * str = String_dup("Value 2");

retCode = stringWriter->write("Key 1", str, HANDLE_NIL);

String_free(str);
```

C++/CLI Example:

```
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

```
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 :

```
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);

String str = "Value 2";

stringWriter.write("Key 1", str, InstanceHandle_t.HANDLE_NIL);
```

3.2.5.3 Keyed String DataReader

The KeyedString *DataReader* API is extended with the following operations (in addition to the standard methods described in Using a Type-Specific DataReader (FooDataReader) (Section 7.4.1)):

```
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 online documentation.

Memory considerations in copy operations:

For read/take operations with copy semantics, such as **read_next_sample()** and **take_next_sample()**, *RTI Data Distribution Service* 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 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:

```
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,
                                         DDS_ANY_SAMPLE_STATE,
                                         DDS_ANY_VIEW_STATE,
                                         DDS_ANY_INSTANCE_STATE);

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));
    }
}
/* Return loan */
retCode = DDS_KeyedStringDataReader_return_loan(
                         stringReader, &data_seq, &info_seq);
```


C++ Example with Namespaces:

```
#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:

```
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,
                   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);
```

3.2.6 Octets Built-in Type

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

C/C++ Representation (without Namespaces):

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

C++/CLI Representation:

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

C# Representation:

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

Java Representation:

```
package com.rti.dds.type.builtin;

public class Bytes implements Copyable {
    public int length;
    public int offset;
    public byte[] value;
    ...
};
```

3.2.6.1 Creating and Deleting Octets

RTI Data Distribution Service provides a set of constructors/destructors to create and destroy Octet objects. For details, select **Modules, DDS API Reference, Topic Module, Built-in Types** in the online documentation.

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

3.2.6.2 Octets DataWriter

In addition to the standard methods (see Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.6)), the octets *DataWriter* API is extended with the following methods:

```
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 online 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:

```
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:

```
#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;
```

C++/CLI Example:

```
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;

octetWriter->write(octets, InstanceHandle_t::HANDLE_NIL);

/* Write some data using individual strings */
array<Byte>^ octetAray = gcnew array<Byte>(1024);
octetArray[0] = 46;
octetArray[1] = 47;
octetsWriter->write(octetArray, 0, 2, InstanceHandle_t::HANDLE_NIL);
```

C# Example:

```
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;

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(octetArray, 0, 2, InstanceHandle_t.HANDLE_NIL);
```

Java Example:

```
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;

octetsWriter.write(octets, InstanceHandle_t.HANDLE_NIL);

/* Write some data using a byte array */
byte[] octetArray = new byte[1024];
octetArray[0] = 46;
octetArray[1] = 47;

octetsWriter.write(octetArray, 0, 2, InstanceHandle_t.HANDLE_NIL);
```

3.2.6.3 Octets DataReader

The octets *DataReader* API matches the standard *DataReader* API (see Using a Type-Specific DataReader (FooDataReader) (Section 7.4.1)). 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()**, *RTI Data Distribution Service* 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:

```
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(&data_seq); ++i) {
    if (DDS_SampleInfoSeq_get_reference(&info_seq, i)->valid_data) {
        DDS_OctetsTypeSupport_print_data(
                         DDS_OctetsSeq_get_reference(&data_seq, i));
    }
}
/* Return loan */
retCode = DDS_OctetsDataReader_return_loan(octetsReader, &data_seq,
                                           &info_seq);
```

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);
```

C# Example:

```
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:

```
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);
```

3.2.7 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/C++ Representation (without Namespaces):

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

C++/CLI Representation:

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

C# Representation:

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

Java Representation:

```
package com.rti.dds.type.builtin;
public class KeyedBytes {
    public String key;
    public int length;
    public int offset;
    public byte[] value;
    ...
};
```

3.2.7.1 Creating and Deleting KeyedOctets

RTI Data Distribution Service provides a set of constructors/destructors to create/destroy KeyedOctets objects. For details, see the online documentation (select **Modules, 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()**, described in the online documentation (select **Modules, 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()**, described in the online documentation (select **Modules, DDS API Reference, Infrastructure Module, String Support**).

3.2.7.2 Keyed Octets DataWriter

In addition to the standard methods (see Using a Type-Specific DataWriter (FooDataWriter) (Section 6.3.6)), the keyed octets *DataWriter* API is extended with the following methods:

```
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,
                    const DDS::Time_t & source_timestamp);

DDS::ReturnCode_t DDS::KeyedOctetsDataWriter::write(
                    const char * key,
                    const unsigned char * octets,
                    int length,
                    const DDS::InstanceHandle_t& handle);

DDS::ReturnCode_t DDS::KeyedOctetsDataWriter::write(
                    const char * key,
                    const DDS::OctetSeq & octets,
                    const DDS::InstanceHandle_t & handle);

DDS::ReturnCode_t DDS::KeyedOctetsDataWriter::write_w_timestamp(
                    const char * key,
                    const unsigned char * octets,
                    int length,
                    const DDS::InstanceHandle_t& handle,
                    const DDS::Time_t& source_timestamp);

DDS::ReturnCode_t DDS::KeyedOctetsDataWriter::write_w_timestamp(
                    const char * key,
                    const DDS::OctetSeq & octets,
                    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 and instance management operations. For more information and a complete description of these operations in all supported languages, see the online 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:

```
DDS_KeyedOctetsDataWriter * octetsWriter = ... ;
DDS_ReturnCode_t retCode;
struct DDS_KeyedOctets * octets = NULL;
char * octetArray = NULL;

/* Write some data using the KeyedOctets structure */
octets = DDS_KeyedOctets_new(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 = DDS_KeyedOctetsDataWriter_write_octets_w_key (
           octetsWriter, "Key 1", octetArray, 2, &DDS_HANDLE_NIL);

free(octetArray);
```

C++ Example with Namespaces:

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

KeyedOctetsDataWriter * octetsWriter = ... ;

/* Write some data using the KeyedOctets structure */
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:

```
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 */
array<Byte>^ octetAray = gcnew array<Byte>(1024);
octetArray[0] = 46;
octetArray[1] = 47;

octetsWriter->write(
          "Key 1", octetArray, 0, 2, InstanceHandle_t::HANDLE_NIL);
```

C# Example:

```
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:

```
import com.rti.dds.publication.*;
import com.rti.dds.type.builtin.*;
import com.rti.dds.infrastructure.*;
...

KeyedBytesDataWriter octetsWriter = ... ;

/* Write some data using the 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;
octetArray[1] = 47;
octetsWriter.write(
           "Key 1", octetArray, 0, 2, InstanceHandle_t.HANDLE_NIL);
```

3.2.7.3 Keyed Octets DataReader

The KeyedOctets *DataReader* API is extended with the following methods (in addition to the standard methods described in Using a Type-Specific DataReader (FooDataReader) (Section 7.4.1)):

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

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

Memory considerations in copy operations:

For read/take operations with copy semantics, such as read_next_sample() and take_next_sample(), *RTI Data Distribution Service* 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 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:

```
struct DDS_KeyedOctetsSeq dataSeq = DDS_SEQUENCE_INITIALIZER;
struct DDS_SampleInfoSeq infoSeq = DDS_SEQUENCE_INITIALIZER;
DDS_KeyedOctetsDataReader * octetsReader = ... ;
DDS_ReturnCode_t retCode;
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:

```
#include "ndds/ndds_namespace_cpp.h"
using namespace DDS;
...
KeyedOctetsSeq dataSeq;
SampleInfoSeq infoSeq;
KeyedOctetsDataReader * 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) {
        KeyedOctetsTypeSupport::print_data(&dataSeq[i]);
    }
}
/* Return loan */
retCode = octetsReader->return_loan(dataSeq, infoSeq);
```

C++/CLI Example:

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

C# Example:

```
using System;
using DDS;
...
KeyedBytesSeq dataSeq = new KeyedButesSeq();
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:

```
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(((KeyedBytes)dataSeq.get(i)).toString());
    }
}
/* Return loan */
octetsReader.return_loan(dataSeq, infoSeq);
```


3.2.8 Managing Memory for Built-in Types

When a 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 sample is received, the *DataReader* deserializes it and stores the result in a sample coming from a pool of preallocated samples.

For data types generated by *rtiddsgen*, the size of the buffers and samples in both pools is known based on the IDL or XML description of the type.

For example:

```
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. It 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/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 PROPERTY QosPolicy (DDS Extension) (Section 6.5.15). Table 3.1 on page 3-47 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*.

Note: These properties must be set consistently with respect to the corresponding ***.max_size** properties in the *DomainParticipant* (see Table 3.13 on page 3-126). 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*.

Section 3.2.8.1 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.*

```
<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**

Built-in Type	Property	Description
string	dds.builtin_type.string.alloc_size	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 3.13 on page 3-126). Otherwise, 1024.
keyed-string	dds.builtin_type.keyed_string.alloc_key_size	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 3.13 on page 3-126). Otherwise, 1024.
	dds.builtin_type.keyed_string.alloc_size	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 3.13 on page 3-126). Otherwise, 1024.
octets	dds.builtin_type.octets.alloc_size	Maximum size of the octet sequences published by the *DataWriter* or *DataReader.* Default: dds.builtin_type.octets.max_size if defined (see Table 3.13 on page 3-126). Otherwise, 2048.
keyed-octets	dds.builtin_type.keyed_octets.alloc_key_size	Maximum size of the key published by the *DataWriter* or received by the *DataReader* (includes the NULL-terminated character). Default: dds.builtin_type.keyed_octets.max_key_size if defined (see Table 3.13 on page 3-126). Otherwise, 1024.
	dds.builtin_type.keyed_octets.alloc_size	Maximum size of the octet sequences published by the *DataWriter* or *DataReader.* Default: dds.builtin_type.keyed_octets.max_size if defined (see Table 3.13 on page 3-126). Otherwise, 2048.

3.2.8.1 Examples—Setting the Maximum Size for a String Programmatically

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

C Example:

```
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);
```

C++ Example with Namespaces:

```
#include "ndds/ndds_namespace_cpp.h"
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);
```

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;
...

Topic stringTopic = ... ;
Publisher publisher = ... ;
DataWriterQos writerQos = new DataWriterQos();

participant.get_default_datawriter_qos(writerQos);

PropertyQosPolicyHelper.add_property (
                      writerQos.property_qos,
                      "dds.builtin_type.string.alloc_size", "1000",
                      false);

StringDataWriter stringWriter =
           (StringDataWriter) publisher.create_datawriter(
                                        stringTopic, writerQos, null,
                                        StatusMask.STATUS_MASK_NONE);
```

Java Example:

```
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, StatusKind.STATUS_MASK_NONE);
```

3.2.9 Type Codes for Built-in Types

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

```
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 3.2.

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

Built-in Type	Property	Description
String	dds.builtin_type.string.max_size	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

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

Built-in Type	Property	Description
Keyed-String	dds.builtin_type.keyed_string.max_key_size	Maximum size of the keys used by the *DataWriters* and *DataReaders* belonging to a *DomainParticipant* (includes the NULL-terminated character). Default: 1024
	dds.builtin_type.keyed_string.max_size	Maximum size of the strings published by the *DataWriters* and received by the *DataReaders* belonging to a *DomainParticipant* using the built-in type (includes the NULL-terminated character). Default: 1024
Octets	dds.builtin_type.octets.max_size	Maximum size of the octet sequences published by the *DataWriters* and *DataReaders* belonging to a *DomainParticipant*. Default: 2048
Keyed-Octets	dds.builtin_type.keyed_octets.max_key_size	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.
	dds.builtin_type.keyed_octets.max_size	Maximum size of the octet sequences published by the *DataWriters* and *DataReaders* belonging to a *DomainParticipant*. Default: 2048

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, C++, C++/CLI, and Java (the languages supported by *rtiddsgen*). The *rtiddsgen* utility parses the IDL file and automatically generates all the necessary routines and wrapper functions to bind the types for use by *RTI Data Distribution Service* at run time.

You will end up with a set of required routines and structures that your application and *RTI Data Distribution Service* will use to manipulate the data.

DDS only uses a subset of the 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 data-centric embedded applications. These include the constructs that define method and function prototypes like "interface."

The *rtiddsgen* utility will parse any file that follows version 3.0.3 of the IDL specification. It will quietly ignore all syntax that is not recognized by DDS. In addition, even though "anonymous sequences" (sequences of sequences with no intervening typedef) are currently legal in IDL, they have been deprecated by the specification, and thus *rtiddsgen* does not support them.

The IDL constructs supported by *rtiddsgen* are described in Table 3.4, "Specifying Data Types in IDL for C and C++," on page 3-57 and Table 3.6, "Specifying Data Types in IDL for Java," on page 3-66. Use these tables to map primitive types to their equivalent IDL syntax, and vice versa.

For C and C++, *rtiddsgen* uses typedefs instead of the language keywords for primitive types. For example, **DDS_Long** instead of **long** or **DDS_Double** instead of **double**. This ensures that the types are of the same size regardless of the platform.[1]

The remainder of this section includes:

- ❏ Variable-Length Types (Section 3.3.1)
- ❏ Value Types (Section 3.3.2)
- ❏ rtiddsgen Translations for IDL Types (Section 3.3.4)
- ❏ Referring to Other IDL Files (Section 3.3.6)
- ❏ Preprocessor Directives (Section 3.3.7)
- ❏ Using Custom Directives (Section 3.3.8)

3.3.1 Variable-Length Types

When *rtiddsgen* generates code for data structures with variable-length types—strings and sequences—it includes functions that create, initialize and finalize (destroy) those

1. The number of bytes sent on the wire for each data type is determined by the Common Data Representation (CDR) standard. For details on CDR, please see the Common Object Request Broker Architecture (CORBA) Specification, Version 3.1, Part 2: CORBA Interoperability, Section 9.3, CDR Transfer Syntax (http://www.omg.org/technology/documents/corba_spec_catalog.htm).

objects. These support functions will properly initialize pointers and allocate and deallocate the memory used for variable-length types. All *RTI Data Distribution Service* 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 sample is written (regardless of whether the type has hard-coded bounds).

3.3.1.1 Sequences

C, C++, C++/CLI, and C# users can allocate memory from a number of sources: from the heap, the stack, or from a custom allocator of some kind. In those languages, 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 online API documentation on *Sequence Support* in the *Infrastructure Module* 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 *rtiddsgen* 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 sample is written will vary.

3.3.1.2 Strings and Wide Strings

The initialization functions that *rtiddsgen* 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 a data structure allocated or initialized by an *RTI Data Distribution Service*-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, *RTI Data Distribution Service* may need to copy a string into a structure that user code has provided. *RTI Data Distribution Service* 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, *RTI Data Distribution Service* 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 *RTI Data Distribution Service* will skip over that string.
- ❏ You must provide a non-NULL string pointer for *RTI Data Distribution Service* to copy into. Otherwise, *RTI Data Distribution Service* will log an error.
- ❏ When you provide a non-NULL string pointer in your data structure, *RTI Data Distribution Service* will copy into the provided memory without performing any additional memory allocations. Be careful—if you provide *RTI Data Distribution Service* with an uninitialized pointer or allocate a string that is too short, you may corrupt the memory or cause a program crash. *RTI Data Distribution Service* 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.

RTI Data Distribution Service 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. *RTI Data Distribution Service* always uses these functions internally for managing string memory; you are recommended—but not required—to use them as well. See the online API documentation on *String Support* in the *Infrastructure Module* for more information about strings.

3.3.2 Value Types

A value type is like a structure, but with support for additional object-oriented features such as inheritance. It is similar to what is sometimes referred to in Java as a *POJO*—a Plain Old Java Object.

Readers familiar with value types in the context of CORBA should consult Table 3.3 to see which value type-related IDL keywords are supported and what their behavior is in the context of *RTI Data Distribution Service*.

Table 3.3 **Value Type Support**

Aspect	Level of Support in rtiddsgen
Inheritance	Single inheritance from other value types
Public state members	Supported
Private state members	Become public when code is generated
Custom keyword	Ignored (the value type is parsed without the keyword and code is generated to work with it)
Abstract value types	No code generated (the value type is parsed, but no code is generated)
Operations	No code generated (the value type is parsed, but no code is generated)
Truncatable keyword	Ignored (the value type is parsed without the keyword and code is generated to work with it)

3.3.3 TypeCode and *rtiddsgen*

Type codes are enabled by default when you run *rtiddsgen*. The **-notypecode** option disables generation of type code information. Type-code support does increase the amount of memory used, so if you need to save on memory, you may consider disabling type codes. (The **-notypecode** option is described in rtiddsgen Command-Line Arguments (Section 3.6.1)

Locally, your application can access the type code for a generated type "Foo" by calling the **Foo::get_typecode()** operation in the code for the type generated by *rtiddsgen* (unless type-code support is disabled with the **-notypecode** option).

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

3.3.4 rtiddsgen Translations for IDL Types

This section describes how to specify your data types in an IDL file. The *rtiddsgen* utility supports all the types listed in the following tables:

- ❏ Table 3.4, “Specifying Data Types in IDL for C and C++,” on page 3-57
- ❏ Table 3.5, “Specifying Data Types in IDL for C++/CLI,” on page 3-63
- ❏ Table 3.6, “Specifying Data Types in IDL for Java,” on page 3-66

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

Table 3.4 **Specifying Data Types in IDL for C and C++**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

char (see Note 1 below)	`struct PrimitiveStruct {`
	`    char char_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_Char char_member;`
		`} PrimitiveStruct;`

wchar	`struct PrimitiveStruct {`
	`    wchar wchar_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_Wchar wchar_member;`
		`} PrimitiveStruct;`

octet	`struct PrimitiveStruct {`
	`    octet octet_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_Octet boolean_member;`
		`} PrimitiveStruct;`

short	`struct PrimitiveStruct {`
	`    short short_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_Short short_member;`
		`} PrimitiveStruct;`

unsigned short	`struct PrimitiveStruct {`
	`    unsigned short`
	`        unsigned_short_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_UnsignedShort`
		`        unsigned_short_member;`
		`} PrimitiveStruct;`

long	`struct PrimitiveStruct {`
	`    long long_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_Long long_member;`
		`} PrimitiveStruct;`

unsigned long	`struct PrimitiveStruct {`
	`    unsigned long`
	`        unsigned_long_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_UnsignedLong`
		`    unsigned_long_member;`
		`} PrimitiveStruct;`

long long	`struct PrimitiveStruct {`
	`    long long long_long_member;`
	`};`
		`typedef struct PrimitiveStruct`
		`{`
		`    DDS_LongLong long_long_member;`
		`} PrimitiveStruct;`

Table 3.4 **Specifying Data Types in IDL for C and C++**

<table>
<tr><th>IDL Type</th><th>Sample Entry in IDL File</th><th>Sample Output Generated by rtiddsgen</th></tr>
<tr><td>unsigned long long</td><td><pre>struct PrimitiveStruct {
    unsigned long long
         unsigned_long_long_member;
};</pre></td><td><pre>typedef struct PrimitiveStruct
{
    DDS_UnsignedLongLong
         unsigned_long_long_member;
} PrimitiveStruct;</pre></td></tr>
<tr><td>float</td><td><pre>struct PrimitiveStruct {
    float float_member;
};</pre></td><td><pre>typedef struct PrimitiveStruct
{
    DDS_Float float_member;
} PrimitiveStruct;</pre></td></tr>
<tr><td>double</td><td><pre>struct PrimitiveStruct {
    double double_member;
};</pre></td><td><pre>typedef struct PrimitiveStruct
{
    DDS_Double double_member;
} PrimitiveStruct;</pre></td></tr>
<tr><td>long double (see Note 2 below)</td><td><pre>struct PrimitiveStruct {
    long double long_double_member;
};</pre></td><td><pre>typedef struct PrimitiveStruct
{
    DDS_LongDouble long_double_member;
} PrimitiveStruct;</pre></td></tr>
<tr><td>pointer (see Note 9 below)</td><td><pre>struct MyStruct {
    long * member;
};</pre></td><td><pre>typedef struct MyStruct {
    DDS_Long * member;
} MyStruct;</pre></td></tr>
<tr><td>boolean</td><td><pre>struct PrimitiveStruct {
    boolean boolean_member;
};</pre></td><td><pre>typedef struct PrimitiveStruct
{
    DDS_Boolean boolean_member;
} PrimitiveStruct;</pre></td></tr>
<tr><td>enum</td><td><pre>enum PrimitiveEnum {
    ENUM1,
    ENUM2,
    ENUM3
};

enum PrimitiveEnum {
    ENUM1 = 10,
    ENUM2 = 20,
    ENUM3 = 30
};</pre></td><td><pre>typedef enum PrimitiveEnum
{
    ENUM1,
    ENUM2,
    ENUM3
} PrimitiveEnum;

typedef enum PrimitiveEnum
{
    ENUM1 = 10,
    ENUM2 = 20,
    ENUM3 = 30
} PrimitiveEnum;</pre></td></tr>
<tr><td>constant</td><td><pre>const short SIZE = 5;</pre></td><td><pre>C: #define SIZE 5
C++: static const DDS_Short size = 5;</pre></td></tr>
</table>

Table 3.4 **Specifying Data Types in IDL for C and C++**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

bitfield
 	
(see Note 12 below)
	struct BitfieldType {
	short myShort_1 : 1;
	unsigned short myUnsignedShort_1: 1;
	long myLong_1: 1;
	unsigned long myUnsignedLong_1 :1;
	char myChar_1 : 1;
	wchar myWChar_1 : 1;
	octet myOctet_1 : 1;
	short : 0;
	long myLong_5 : 5;
	long myLong_30 : 30;
	short myShort_6 : 6;
	short myShort_3and4 : 3+4;
	short myShort;
	short myShort_8 : 8;
	long myLong_32: 32;
	};
		typedef struct BitfieldType
		{
		DDS_Short myShort_1 : 1;
		DDS_UnsignedShort myUnsignedShort_1 : 1;
		DDS_Long myLong_1 : 1;
		DDS_UnsignedLong myUnsignedLong_1 : 1;
		DDS_Char myChar_1 : 1;
		DDS_Wchar myWChar_1 : 1;
		DDS_Octet myOctet_1 : 1;
		DDS_Short : 0;
		DDS_Long myLong_5 : 5;
		DDS_Long myLong_30 : 30;
		DDS_Short myShort_6 : 6;
		DDS_Short myShort_3and4 : 3+4;
		DDS_Short myShort;
		DDS_Short myShort_8 : 8;
		DDS_Long myLong_32 : 32;
		} BitfieldType;

struct
 	
(see Note 10 below)
	struct PrimitiveStruct {
	char char_member;
	};
		typedef struct PrimitiveStruct
		{
		char char_member;
		} PrimitiveStruct;

union
 	
(see Note 3 and Note 10 below)
	union PrimitiveUnion switch (long){
	case 1:
	short short_member;
	default:
	long long_member;
	};
		typedef struct PrimitiveUnion
		{
		DDS_Long _d;
		struct {
		DDS_Short short_member;
		DDS_Long long_member;
		} _u;
		} PrimitiveUnion;

typedef	typedef short TypedefShort;	typedef DDS_Short TypedefShort;

array of above types	struct OneDArrayStruct {
	short short_array[2];
	};
	
	struct TwoDArrayStruct {
	short short_array[1][2];
	};
		typedef struct OneDArrayStruct
		{
		DDS_Short short_array[2];
		} OneDArrayStruct;
		
		typedef struct TwoDArrayStruct
		{
		DDS_Short short_array[1][2];
		} TwoDArrayStruct;

Table 3.4 **Specifying Data Types in IDL for C and C++**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

bounded sequence of above types
 	
(see Note 11 below)
	`struct SequenceStruct {`
	`   sequence<short,4> short_sequence;`
	`};`
		`typedef struct SequenceStruct`
		`{`
		`    DDSShortSeq short_sequence;`
		`} SequenceStruct;`
		
		Note: Sequences of primitive types have been predefined by *RTI Data Distribution Service*.

unbounded sequence of above types
 	
(see Note 11 below)
	`struct SequenceStruct {`
	`   sequence<short> short_sequence;`
	`};`
		`typedef struct SequenceStruct`
		`{`
		`    DDSShortSeq short_sequence;`
		`} SequenceStruct;`
		
		Note: *rtiddsgen* will supply a default bound. You can specify that bound with the "-sequenceSize" command-line option; see Section 3.6.1.

array of sequences	`struct ArraysOfSequences{`
	`   sequence<short,4>`
	`       sequences_array[2];`
	`};`
		`typedef struct ArraysOfSequences`
		`{`
		`    DDS_ShortSeq sequences_array[2];`
		`} ArraysOfSequences;`

sequence of arrays
 	
(see Note 11 below)
	`typedef short ShortArray[2];`
	
	`struct SequenceofArrays {`
	`   sequence<ShortArray,2>`
	`       arrays_sequence;`
	`};`
		`typedef DDS_Short ShortArray[2];`
		
		`DDS_SEQUENCE_NO_GET(ShortArraySeq,`
		`                    ShortArray);`
		
		`typedef struct SequenceOfArrays`
		`{`
		`    ShortArraySeq arrays_sequence;`
		`} SequenceOfArrays;`
		
		DDS_SEQUENCE_NO_GET is an *RTI Data Distribution Service* macro that defines a new sequence type for a user data type. In this case, the user data type is ShortArray.

sequence of sequences
 	
(see Note 4 and Note 11 below)
	`typedef sequence<short,4>`
	`    ShortSequence;`
	
	`struct SequencesOfSequences{`
	`    sequence<ShortSequence,2>`
	`        sequences_sequence;`
	`};`
		`typedef DDS_ShortSeq ShortSequence;`
		
		`DDS_SEQUENCE(ShortSequenceSeq,`
		`             ShortSequence);`
		
		`typedef struct SequencesOfSequences{`
		`    ShortSequenceSeq`
		`    sequences_sequence;`
		`} SequencesOfSequences;`

Table 3.4 **Specifying Data Types in IDL for C and C++**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

bounded string	struct PrimitiveStruct {
	    string<20> string_member;
	};
		typedef struct PrimitiveStruct {
		    char* string_member;
		            /* maximum length = (20) */
		} PrimitiveStruct;

unbounded string	struct PrimitiveStruct {
	    string string_member;
	};
		typedef struct PrimitiveStruct {
		    char* string_member;
		            /* maximum length = (255) */
		} PrimitiveStruct;
		
		Note: rtiddsgen will supply a default bound. You can specify that bound with the **-stringSize** command-line option, see Section 3.6.1.

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;
		
		Note: *rtiddsgen* will supply a default bound.

module	module PackageName {
	    struct Foo {
	        long field;
	    };
	};
		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;

Table 3.4 **Specifying Data Types in IDL for C and C++**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

valuetype
 	
(see Note 9 and Note 10 below)
	`valuetype MyValueType {`
	`   public MyValueType2 * member;`
	`};`
	
	`valuetype MyValueType {`
	`   public MyValueType2 member;`
	`};`
	
	`valuetype MyValueType: MyBaseValueType`
	`    {`
	`   public MyValueType2 * member;`
	`};`
		C++: `class MyValueType {`
		`public:`
		`   MyValueType2 * member;`
		`};`
		
		`class MyValueType {`
		`public:`
		`   MyValueType2  member;`
		`};`
		
		`class MyValueType : public MyBa-`
		`seValueType`
		`{`
		`public:`
		`   MyValueType2 * member;`
		`};`
		
		C: `typedef struct MyValueType {`
		`    MyValueType2 * member;`
		`} MyValueType;`
		
		`typedef struct MyValueType {`
		`    MyValueType2  member;`
		`} MyValueType;`
		
		`typedef struct MyValueType`
		`{`
		`    MyBaseValueType parent;`
		`    MyValueType2 * member;`
		`} MyValueType;`

Table 3.5 **Specifying Data Types in IDL for C++/CLI**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

char (see Note 1 below)	`struct PrimitiveStruct {`
	`    char char_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::Char char_member;`
		`};`

wchar	`struct PrimitiveStruct {`
	`    wchar wchar_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::Char wchar_member;`
		`};`

octet	`struct PrimitiveStruct {`
	`    octet octet_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::Byte octet_member;`
		`};`

short	`struct PrimitiveStruct {`
	`    short short_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::Int16 short_member;`
		`};`

unsigned short	`struct PrimitiveStruct {`
	`    unsigned short`
	`        unsigned_short_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`  System::UInt16 unsigned_short_member;`
		`};`

long	`struct PrimitiveStruct {`
	`    long long_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`   System::Int32 long_member;`
		`};`

unsigned long	`struct PrimitiveStruct {`
	`   unsigned long unsigned_long_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`   System::UInt32 unsigned_long_member;`
		`};`

long long	`struct PrimitiveStruct {`
	`    long long long_long_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::Int64 long_long_member;`
		`};`

unsigned long long	`struct PrimitiveStruct {`
	`unsigned long long`
	`    unsigned_long_long_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::UInt64`
		`        unsigned_long_long_member;`
		`};`

float	`struct PrimitiveStruct {`
	`    float float_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::Single float_member;`
		`};`

double	`struct PrimitiveStruct {`
	`    double double_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::Double double_member;`
		`} PrimitiveStruct;`

long double (see Note 2 below)	`struct PrimitiveStruct {`
	`    long double long_double_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`   DDS::LongDouble long_double_member;`
		`} PrimitiveStruct;`

Table 3.5 **Specifying Data Types in IDL for C++/CLI**

<table>
<tr><th>IDL Type</th><th>Sample Entry in IDL File</th><th>Sample Output Generated by rtiddsgen</th></tr>
<tr><td>boolean</td><td><code>struct PrimitiveStruct {
    boolean boolean_member;
};</code></td><td><code>public ref class PrimitiveStruct {
    System::Boolean boolean_member;
};</code></td></tr>
<tr><td>enum</td><td><code>enum PrimitiveEnum {
    ENUM1,
    ENUM2,
    ENUM3
};

enum PrimitiveEnum {
    ENUM1 = 10,
    ENUM2 = 20,
    ENUM3 = 30
};</code></td><td><code>public enum class
PrimitiveEnum : System::Int32 {
    ENUM1,
    ENUM2,
    ENUM3
};

public enum class
PrimitiveEnum : System::Int32 {
    ENUM1 = 10,
    ENUM2 = 20,
    ENUM3 = 30
};</code></td></tr>
<tr><td>constant</td><td><code>const short SIZE = 5;</code></td><td><code>public ref class SIZE {
    public:
        static System::Int16 VALUE = 5;
};</code></td></tr>
<tr><td>struct

(see Note 10 below)</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>union

(see Note 3 and Note 10 below)</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;
    struct PrimitiveUnion_u {
        System::Int16 short_member;
        System::Int32 long_member;
    } _u;
};</code></td></tr>
<tr><td>array of above types</td><td><code>struct OneDArrayStruct {
    short short_array[2];
};</code></td><td><code>public ref class OneDArrayStruct {
    array<System::Int16>^ short_array;
    /*length == 2*/
};</code></td></tr>
</table>

Table 3.5 **Specifying Data Types in IDL for C++/CLI**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

bounded sequence of above types
 	
(see Note 11 below)
	`struct SequenceStruct {`
	`   sequence<short,4> short_sequence;`
	`};`
		`public ref class SequenceStruct {`
		`    ShortSeq^ short_sequence;`
		`    /*max = 4*/`
		`};`
		
		Note: Sequences of primitive types have been predefined by *RTI Data Distribution Service*.

unbounded sequence of above types
 	
(see Note 11 below)
	`struct SequenceStruct {`
	`   sequence<short> short_sequence;`
	`};`
		`public ref class SequenceStruct {`
		`    ShortSeq^ short_sequence;`
		`    /*max = <default bound>*/`
		`};`
		Note: *rtiddsgen* will supply a default bound. You can specify that bound with the **-sequenceSize** command-line option; see Section 3.6.1.

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 {`
	`    string<20> 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)`
		`};`
		Note: *rtiddsgen* will supply a default bound. You can specify that bound with the **-stringSize** command-line option, see Section 3.6.1.

bounded wstring	`struct PrimitiveStruct {`
	`    wstring<20> wstring_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::String^ string_member;`
		`    // maximum length = (20)`
		`};`

3. User Data Types

Table 3.5 **Specifying Data Types in IDL for C++/CLI**

IDL Type	Sample Entry in IDL File	Sample Output Generated by rtiddsgen

unbounded wstring	`struct PrimitiveStruct {`
	`    wstring wstring_member;`
	`};`
		`public ref class PrimitiveStruct {`
		`    System::String^ string_member; //`
		`    maximum length = (255)`
		`};`
		Note: *rtiddsgen* will supply a default bound. You can specify that bound with the **-stringSize** command-line option, see Section 3.6.1.

module	`module PackageName {`
	`    struct Foo {`
	`        long field;`
	`    };`
	`};`
		`namespace PackageName {`
		`    public ref class Foo {`
		`        System::Int32 field;`
		`    };`
		`};`

Table 3.6 **Specifying Data Types in IDL for Java**

IDL Type	Sample Entry in IDL file	Sample Java Output Generated by rtiddsgen

char
 	
(see Note 5 below)
	`struct PrimitiveStruct {`
	`    char char_member;`
	`};`
		`public class PrimitiveStruct`
		`{`
		`    public char char_member;`
		`    ...`
		`}`

wchar
 	
(see Note 5 below)
	`struct PrimitiveStruct {`
	`    wchar wchar_member;`
	`};`
		`public class PrimitiveStruct`
		`{`
		`    public char wchar_member;`
		`    ...`
		`}`

octet	`struct PrimitiveStruct {`
	`    octet octet_member;`
	`};`
		`public class PrimitiveStruct`
		`{`
		`    public byte byte_member;`
		`    ...`
		`}`

short	`struct PrimitiveStruct {`
	`    short short_member;`
	`};`
		`public class PrimitiveStruct`
		`{`
		`    public short short_member;`
		`    ...`
		`}`

Table 3.6 **Specifying Data Types in IDL for Java**

<table>
<tr><th>IDL Type</th><th>Sample Entry in IDL file</th><th>Sample Java Output Generated by rtiddsgen</th></tr>
<tr><td>unsigned short

(see Note 6 below)</td><td><pre>struct PrimitiveStruct {
    unsigned short
       unsigned_short_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public short unsigned_short_member;
    ...
}</pre></td></tr>
<tr><td>long</td><td><pre>struct PrimitiveStruct {
    long long_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public int long_member;
    ...
}</pre></td></tr>
<tr><td>unsigned long

(see Note 6 below)</td><td><pre>struct PrimitiveStruct {
    unsigned long
       unsigned_long_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public int unsigned_long_member;
    ...
}</pre></td></tr>
<tr><td>long long</td><td><pre>struct PrimitiveStruct {
    long long long_long_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public long long_long_member;
    ...
}</pre></td></tr>
<tr><td>unsigned long long

(see Note 7 below)</td><td><pre>struct PrimitiveStruct {
    unsigned long long
        unsigned_long_long_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public long
        unsigned_long_long_member;
    ...
}</pre></td></tr>
<tr><td>float</td><td><pre>struct PrimitiveStruct {
    float float_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public float float_member;
    ...
}</pre></td></tr>
<tr><td>double</td><td><pre>struct PrimitiveStruct {
    double double_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public double double_member;
    ...
}</pre></td></tr>
<tr><td>long double

(see Note 7 below)</td><td><pre>struct PrimitiveStruct {
    long double long_double_member;
};</pre></td><td><pre>public class PrimitiveStruct
{
    public double long_double_member;
    ...
}</pre></td></tr>
</table>

Table 3.6 **Specifying Data Types in IDL for Java**

IDL Type	Sample Entry in IDL file	Sample Java Output Generated by rtiddsgen

pointer
(see Note 9 below)
	struct MyStruct {
	long * member;
	};
		public class MyStruct {
		public int member;
		...
		};

boolean	struct PrimitiveStruct {
	boolean boolean_member;
	};
		public class PrimitiveStruct
		{
		public boolean boolean_member;
		...
		}

enum	enum PrimitiveEnum {
	ENUM1,
	ENUM2,
	ENUM3
	};
		public class PrimitiveEnum extends Enum
		{
		public static PrimitiveEnum ENUM1 =
		new PrimitiveEnum ("ENUM1", 0);
		
		public static PrimitiveEnum ENUM2 =
		new PrimitiveEnum ("ENUM2", 1);
		
		public static PrimitiveEnum ENUM3 =
		new PrimitiveEnum ("ENUM3", 2);
		
		public static PrimitiveEnum
		valueOf(int ordinal);
		
		...
		}

	enum PrimitiveEnum {
	ENUM1 = 10,
	ENUM2 = 20,
	ENUM3 = 30
	};
		public class PrimitiveEnum extends Enum
		{
		public static PrimitiveEnum ENUM1 =
		new PrimitiveEnum ("ENUM1", 10);
		
		public static PrimitiveEnum ENUM2 =
		new PrimitiveEnum ("ENUM2", 10);
		
		public static PrimitiveEnum ENUM3 =
		new PrimitiveEnum ("ENUM3", 20);
		
		public static PrimitiveEnum
		valueOf(int ordinal);
		
		...
		}

Table 3.6 **Specifying Data Types in IDL for Java**

IDL Type	Sample Entry in IDL file	Sample Java Output Generated by rtiddsgen

constant	`const short SIZE = 5;`	`public class SIZE {`
		`    public static final short VALUE = 5;`
		`}`

bitfield
 	
(see Note 12 below)
	`struct BitfieldType {`
	`      short myShort_1 : 1;`
	`      long myLong_1: 1;`
	`      char myChar_1 : 1;`
	`      wchar myWChar_1 : 1;`
	`      octet myOctet_1 : 1;`
	`      short : 0;`
	`      long myLong_5 : 5;`
	`      long myLong_30 : 30;`
	`      short myShort_6 : 6;`
	`      short myShort_3and4 : 3+4;`
	`      short myShort;`
	`      short myShort_8 : 8;`
	`      long myLong_32: 32;`
	`};`
		`public class BitfieldType`
		`{`
		`    public short myShort_1;`
		`    public int myLong_1;`
		`    public byte myChar_1;`
		`    public char myWChar_1;`
		`    public byte myOctet_1;`
		`    public int myLong_5;`
		`    public int myLong_30;`
		`    public short myShort_6;`
		`    public short myShort_3and4;`
		`    public short myShort;`
		`    public short myShort_8;`
		`    public int myLong_32;`
		`    ...`
		`}`

struct
 	
(see Note 10 below)
	`struct PrimitiveStruct {`
	`   char char_member;`
	`};`
		`public class PrimitiveStruct`
		`{`
		`    public char char_member;`
		`}`

union
 	
(see Note 10 below)
	`union PrimitiveUnion switch (long){`
	`    case 1:`
	`        short short_member;`
	`    default:`
	`        long long_member;`
	`};`
		`public class PrimitiveUnion {`
		`    public int _d;`
		`    public short short_member;`
		`    public int long_member;`
		`    ...`
		`}`

typedef of primitives, enums, strings
 	
(see Note 8 below)
	`typedef short ShortType;`
	
	`struct PrimitiveStruct {`
	`    ShortType short_member;`
	`};`
		`/* typedefs are unwounded to the original`
		`   type when used */`
		`public class PrimitiveStruct`
		`{`
		`     public short short_member;`
		`    ...`
		`}`

Table 3.6 **Specifying Data Types in IDL for Java**

IDL Type	Sample Entry in IDL file	Sample Java Output Generated by rtiddsgen

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];
		...
		}

	struct TwoDArrayStruct {
	short short_array[1][2];
	};
		public class TwoDArrayStruct
		{
		public short[][] short_array = new
		short[1][2];
		...
		}

bounded sequence
 	
(see Note 11 below)
	struct SequenceStruct {
	sequence<short,4>
	short_sequence;
	};
		public class SequenceStruct
		{
		public ShortSeq short_sequence = new
		ShortSeq((4));
		...
		}
		Note: Sequences of primitive types have been pre-defined by *RTI Data Distribution Service*.

unbounded sequence
 	
(see Note 11 below)
	struct SequenceStruct {
	sequence<short> short_sequence;
	};
		public class SequenceStruct
		{
		public ShortSeq short_sequence = new
		ShortSeq((100));
		...
		}
		Note: *rtiddsgen* will supply a default bound. You can specify that bound with the "-sequenceSize" command-line option; see Section 3.6.1.

array of sequences	struct ArraysOfSequences{
	sequence<short,4>
	sequences_array[2];
	};
		public class ArraysOfSequences
		{
		public ShortSeq[] sequences_array =
		new ShortSeq[2];
		...
		}

Table 3.6 **Specifying Data Types in IDL for Java**

<table>
<tr><th>IDL Type</th><th>Sample Entry in IDL file</th><th>Sample Java Output Generated by rtiddsgen</th></tr>
<tr><td>sequence of arrays

(see Note 11 below)</td><td><pre>typedef short ShortArray[2];

struct SequenceOfArrays{
    sequence<ShortArray,2>
        arrays_sequence;
};</pre></td><td><pre>/* Wrapper class */
public class ShortArray
{
    public short[] userData = new
        short[2];
    ...
}

/* Sequence of wrapper class objects */
public final class ShortArraySeq
    extends ArraySequence
{
    ...
}

public class SequenceOfArrays
{
    public ShortArraySeq arrays_sequence
        = new ShortArraySeq((2));
    ...
}</pre></td></tr>
<tr><td>sequence of sequences

(see Note 4 and Note 11 below)</td><td><pre>typedef sequence<short,4>
    ShortSequence;

struct SequencesOfSequences{
    sequence<ShortSequence,2>
        sequences_sequence;
};</pre></td><td><pre>/* Wrapper class */
public class ShortSequence
{
    public ShortSeq userData = new
        ShortSeq((4));
    ...
}

/* Sequence of wrapper class objects */
public final class ShortSequenceSeq
    extends ArraySequence
{
    ...
}

public class SequencesOfSequences
{
    public ShortSequenceSeq
        sequences_sequence = new
            ShortSequenceSeq((2));
    ...
}</pre></td></tr>
</table>

Table 3.6 **Specifying Data Types in IDL for Java**

IDL Type	Sample Entry in IDL file	Sample Java Output Generated by rtiddsgen

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) */
		    ...
		}
		Note: *rtiddsgen* will supply a default bound. You can specify that bound with the **-stringSize** command-line option, see Section 3.6.1.

bounded wstring	struct PrimitiveStruct {
	    wstring<20> wstring_member;
	};
		public class PrimitiveStruct
		{
		    public String wstring_member = new
		        String();
		        /* maximum length = (20) */
		    ...
		}

unbounded wstring	struct PrimitiveStruct {
	    wstring wstring_member;
	};
		public class PrimitiveStruct
		{
		    public String wstring_member = new
		        String();
		        /* maximum length = (255) */
		    ...
		}
		Note: *rtiddsgen* will supply a default bound.

module	module PackageName {
	    struct Foo {
	        long field;
	    };
	};
		package PackageName;
		
		public class Foo
		{
		    public int field;
		    …
		}

Table 3.6 **Specifying Data Types in IDL for Java**

IDL Type	Sample Entry in IDL file	Sample Java Output Generated by rtiddsgen

valuetype
 	
(see Note 9 and Note 10 below)
	`valuetype MyValueType {`
	`   public MyValueType2 * member;`
	`};`
	
	`valuetype MyValueType {`
	`   public MyValueType2 member;`
	`};`
	
	`valuetype MyValueType:`
	`    MyBaseValueType  {`
	`   public MyValueType2 * member;`
	`};`
		`public class MyValueType  {`
		`    public MyValueType2 member;`
		`    ….`
		`};`
		
		`public class MyValueType  {`
		`    public MyValueType2 member;`
		`    ….`
		`};`
		
		`public class MyValueType extends`
		`    MyBaseValueType`
		`{`
		`    public MyValueType2 member;`
		`    ….`
		`}`

Notes for Table 3.4 through Table 3.6:

Note 1. Note that 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 *RTI Data Distribution Service* to use the native type instead. That is, if **sizeof(long double) == 16**, you can tell *RTI Data Distribution Service* to map **DDS_LongDouble** to **long 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 and C++, so that *RTI Data Distribution Service* 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, *RTI Data Distribution Service* 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. Such sequences are deprecated in CORBA and may be removed from future versions of IDL. For example, this is *not* supported:

Not Supported—>

```
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:

Supported—>

```
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. There are no unsigned types in Java. 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, *rtiddsgen* will generate wrapper classes.

Note 9. In C and C++, all the members in a value type, structure or union that are declared with the pointer symbol ('*') will be mapped to references (pointers). In C++/CLI and Java, the pointer symbol is ignored because the members are always mapped as references.

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

Not Supported—>

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

Note 11. The DDS implied type **<Type>Seq** cannot be explicitly declared in the IDL file. For example, this is *not* supported:

Not Supported—>

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

Note 12. Data types containing bitfield members are not supported by DynamicData (Section 3.8).

3.3.5 Escaped Identifiers

To use an IDL keyword as an identifier, the keyword has to be "escaped" by prepending an underscore, '_'. In addition, you must run *rtiddsgen* with the **-enableEscapeChar** option. For example:

```
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:

```
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:

```
struct MyStruct {
    long int; // error
    long _int; // error
};
```

3.3.6 Referring to Other IDL Files

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

```
#include "Bar.idl"
```

If such a statement is encountered by *rtiddsgen* and you are generating code for C, C++, and C++/CLI, *rtiddsgen* will assume that code has been generated for **Bar.idl** with corresponding header files, **Bar.h** and **BarPlugin.h**.

The generated code will automatically have:

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

added where needed to compile correctly.

Because Java types do not refer to one another in the same way, it is not possible for *rtiddsgen* to automatically generate Java **import** statements based on an IDL **#include** statement. Any **#include** statements will be ignored when Java code is generated. To add imports to your generated Java code, you should use the **//@copy** directive (see Section 3.3.8.2).

3.3.7 Preprocessor Directives

rtiddsgen supports the standard preprocessor directives defined by the IDL specification, such as **#if**, **#endif**, **#include**, and **#define**.

To support these directives, *rtiddsgen* 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 rtiddsgen Command-Line Arguments (Section 3.6.1).

3.3.8 Using Custom Directives

The following *rtiddsgen*-specific directives can be used in your IDL file:

//@key (see Section 3.3.8.1)

//@copy (see Section 3.3.8.2)
//@copy-c
//@copy-cppcli
//@copy-java
//@copy-declaration

//@copy-c-declaration
//@copy-cppcli-declaration
//@copy-java-declaration

//@resolve-name [true | false] (see Section 3.3.8.3)

//@top-level [true | false] (see Section 3.3.8.4)

Custom directives start with "//@". Note: Do not put a space between the slashes and the @, or the directive will not be recognized by *rtiddsgen*.

The directives are also case-sensitive. For instance, you must use **//@key** (not **//@Key**).

3.3.8.1 The @key Directive

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

With each key, *RTI Data Distribution Service* 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, *RTI Data Distribution Service* 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** definitions in IDL may have key fields. When *rtiddsgen* encounters **//@key**, it considers the previously declared field in the enclosing structure to be part of the key. Table 3.7 on page 3-77 shows some examples of keys.

Table 3.7 **Example Keys**

Type	Key Fields

`struct NoKey {`
`    long member1;`
`    long member2;`
`}`	

`struct SimpleKey {`
`    long member1; //@key`
`    long member2;`
`}`
	`member1`

`struct NestedKey {`
`    SimpleKey member1; //@key`
`    long member2;`
`}`
	`member1.member1`

Table 3.7 **Example Keys**

Type	Key Fields

`struct NestedKey2 {`
`    NoKey member1; //@key`
`    long member2;`
`}`
	`member1.member1`
	`member1.member2`

`valuetype BaseValueKey {`
`    public long member1; //@key`
`}`
	`member1`

`valuetype DerivedValueKey :BaseValueKey {`
`    public long member2; //@key`
`}`
	`member1`
	`member2`

`valuetype DerivedValue : BaseValueKey {`
`    public long member2;`
`}`
	`member1`

`struct ArrayKey {`
`    long member1[3]; //@key`
`}`
	`member1[0]`
	`member1[1]`
	`member1[2]`

3.3.8.2 The @copy and Related Directives

To copy a line of text verbatim into the generated code files, use the **@copy** directive in the IDL file. This feature 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.

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

For code that should only be copied if the language is C or C++, use **@copy-c**. For code that should only be copied if the language is C++/CLI, use **@copy-cppcli**. For code that should only be copied if the language is Java, use **@copy-java**. These variations allow you to use the same IDL file for multiple languages. For example, to add import statements to generated Java code:

```
//@copy-java import java.util.*;
```

The above line would be ignored if the same IDL file was used to generate C, C++, and C++/CLI 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.

If you want *rtiddsgen* 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 directive.

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.

//@copy-declaration only copies the text into the file where the type is declared (<type>.h for C and C++, or <type>.java for Java).

//@copy-c-declaration is the same as **//@copy-declaration**, but is for C and C++ code.

//@copy-cppcli-declaration is the same as **//@copy-declaration**, but is for C++/CLI code.

//@copy-java-declaration is the same as **//@copy-declaration**, but is for Java-only code.

3. User Data Types

3.3.8.3 The @resolve-name Directive

In IDL, the "**module**" keyword is used to create namespaces for the declaration of types and classes defined within the file. Here is an example IDL definition:

```
module PackageName {
    struct Foo {
        long field;
    };
};
```

For C++ and C++/CLI, you may use the **-namespace** command-line option, which causes *rtiddsgen* to generate a namespace, such as the following:

```
namespace PackageName{
   typedef struct Foo {
      DDS_Long field;
   } Foo;
} PackageName;
```

When generating C++/CLI, the **-namespace** option is considered to always be passed. Module names are never prepended to class names.

For C, or if you do not use the **-namespace** command-line option for C++ or C++/CLI, the name of the module is concatenated with the name of the structure to create the namespace. The resulting code looks like this:

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

In Java, 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:

```
public class Foo {
    public int field;
    ...
};
```

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

```
module PackageName {
    struct Bar {
        long field;
    };
    struct Foo {
        Bar  barField;
    };
};
```

When *rtiddsgen* 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.

In C or C++, if you do not use **-namespace**, the resulting code will be:

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

typedef struct PackageName_Foo {
    PackageName_Bar barField;
} PackageName_Foo;
```

In C++, if you use **-namespace**, the resulting code will be:

```
namespace PackageName {
    typedef struct Bar {
        DDS_Long  field;
    } Bar;

    typedef struct Foo
    {
        PackageName::Bar  barField;
    } Foo;
}
```

And in Java, **PackageName/Bar.java** and **PackageName/Foo.java** would be created with the following code respectively:

```
public class Bar {
    public int field;
    ...
};
```

and

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

However, sometimes you may not want *rtiddsgen* to resolve the types of variables when modules are used. In the example above, instead of referring to the **Bar** as defined by the same package, you may want the **barField** in **Foo** to use **Bar** directly without prepending a module name. To specify that *rtiddsgen* should not resolve the scope of a type, use the '**//@resolve-name false**' directive.

For example:

```
module PackageName {
    struct Bar {
        long field;
    };

    struct Foo {
        Bar  barField;  //@resolve-name false
    };
};
```

When this directive is used, then for the field preceding the directive, *rtiddsgen* respects the resolution of its type name indicated in the IDL file. It will use the type unmodified in the generated code. In C and C++:

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

typedef struct PackageName_Foo {
    Bar barField;
} PackageName_Foo;
```

And in Java, in **PackageName/Bar.java** and **PackageName/Foo.java** respectively:

```
public class Bar {
    public int field;
    ...
};
```

and

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

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.

When used at the end of the declaration of a structure in IDL, then the directive applies to all types within the structure.

```
struct MyStructure {
    Foo member1;
   Bar member2;
}; //@resolve-name false
```

By default, without using the directive, *rtiddsgen* will try to resolve the type of a field and to use the fully qualified name in the generated code. If the type is not found to be defined within the same scope as the structure in which it is used or in a parent scope, then *rtiddsgen* will generate code with just the type name itself, assuming that the name will be resolved by the compiler through other means available to the user (header files or import statements). A type is in the same scope as the structure if both the type and the structure in which it is used are defined within the same module.

3.3.8.4 The @top-level Directive

By default, *rtiddsgen* 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 the DDS API 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, *rtiddsgen* 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 directive '**//@top-level false**' to tell *rtiddsgen* 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.

In this example, *rtiddsgen* will generate *DataWriter/DataReader* code for **TopLevelStruct** only:

```
struct EmbeddedStruct{
    short member;
}; //@top-level false

struct TopLevelStruct{
    EmbeddedStruct member;
};
```

3.4 Creating User Data Types with Extensible Markup Language (XML)

You can describe user data types with Extensible Markup Language (XML) notation. *RTI Data Distribution Service* provides DTD and XSD files that describe the XML format; see **<NDDSHOME>/resource/qos_profiles_4.5x/rtiddsgen/schema/rti_dds_topic_types.dtd** and **<NDDSHOME>/resource/qos_profiles_4.5x/rtiddsgen/**

3. User Data Types

schema/rti_dds_topic_types.xsd, respectively (in 4.5*x*, the *x* stands for the version letter of the current release).

The XML validation performed by *rtiddsgen* always uses the DTD definition. If the <!DOCTYPE> tag is not in the XML file, *rtiddsgen* will look for the default DTD document in **<NDDSHOME>/resource/rtiddsgen/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[1]:

```
<?xml version="1.0" encoding="UTF-8"?>
    <types xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:noNamespaceSchemaLocation=
  "<same as NDDSHOME>/resource/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[1]:

```
<?xml version="1.0" encoding="UTF-8"?>
      <!DOCTYPE types SYSTEM
    "<same as NDDSHOME>/resource/rtiddsgen/schema/rti_dds_topic_types.dtd">
      <types>
          ...
      </types>
```

Table 3.8 shows how to map the type system constructs into XML.

1. Replace <*same as NDDSHOME*> with the full path to the *RTI Data Distribution Service* installation directory.

Table 3.8 **Mapping Type System Constructs to XML**

Type/Construct		Example	
IDL	**XML**	**IDL**	**XML**

char	char	struct PrimitiveStruct {
		  char char_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="char_member"
			   type="char"/>
			</struct>

wchar	wchar	struct PrimitiveStruct {
		  wchar wchar_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="wchar_member"
			   type="wchar"/>
			</struct>

octet	octet	struct PrimitiveStruct {
		  octet octet_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="octet_member"
			   type="octet"/>
			</struct>

short	short	struct PrimitiveStruct {
		  short short_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="short_member"
			   type="short"/>
			</struct>

unsigned short	unsignedShort	struct PrimitiveStruct {
		  unsigned short
		    unsigned_short_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="unsigned_short_member"
			  type="unsignedShort"/>
			</struct>

long	long	struct PrimitiveStruct {
		  long long_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="long_member"type="long"/>
			</struct>

unsigned long	unsignedLong	struct PrimitiveStruct {
		  unsigned long
		     unsigned_long_member;
		};
			<struct name="PrimitiveStruct">
			   <member name= "unsigned_long_member"
			   type="unsignedLong"/>
			</struct>

long long	longLong	struct PrimitiveStruct {
		   long long
		        long_long_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="long_long_member"
			   type="longLong"/>
			</struct>

unsigned long long	unsignedLongLong	struct PrimitiveStruct {
		 unsigned long long
		unsigned_long_long_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="unsigned_long_long_member"
			   type="unsignedLongLong"/>
			</struct>

float	float	struct PrimitiveStruct {
		  float float_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="float_member"
			   type="float"/>
			</struct>

Table 3.8 **Mapping Type System Constructs to XML**

Type/Construct		Example	
IDL	**XML**	**IDL**	**XML**

double	double	struct PrimitiveStruct {
		  double double_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="double_member"
			   type="double"/>
			</struct>

long double	longDouble	struct PrimitiveStruct {
		  long double
		     long_double_member;
		};
			<struct name="PrimitiveStruct">
			  <member name= "long_double_member"
			   type="longDouble"/>
			</struct>

boolean	boolean	struct PrimitiveStruct {
		  boolean boolean_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="boolean_member"
			   type="boolean"/>
			</struct>

unbounded string	string without stringMaxLength attribute or with stringMaxLength set to -1	struct PrimitiveStruct {
		  string string_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="string_member"
			   type="string"/>
			</struct>
			or
			<struct name="PrimitiveStruct">
			  <member name="string_member"
			   type="string" stringMaxLength="-1"/>
			</struct>

bounded string	string with stringMaxLength attribute	struct PrimitiveStruct {
		  string<20> string_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="string_member"
			  type="string" stringMaxLength="20"/>
			</struct>

unbounded wstring	wstring without stringMaxLength attribute or with stringMaxLength set to -1	struct PrimitiveStruct {
		  wstring wstring_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="wstring_member"
			   type="wstring"/>
			</struct>
			or
			<struct name="PrimitiveStruct">
			  <member name="wstring_member"
			   type="wstring" stringMaxLength="-1"/>
			</struct>

bounded wstring	wstring with stringMaxLength attribute	struct PrimitiveStruct {
		  wstring<20>
		     wstring_member;
		};
			<struct name="PrimitiveStruct">
			  <member name="wstring_member"
			 type="wstring" stringMaxLength="20"/>
			</struct>

Table 3.8 **Mapping Type System Constructs to XML**

Type/Construct		Example	
IDL	**XML**	**IDL**	**XML**

pointer	pointer attribute with values true,false,0 or 1
	Default (if not present): 0
		`struct PrimitiveStruct {`
		`  long * long_member;`
		`};`
			`<struct name="PointerStruct">`
			`  <member name="long_member" type="long"`
			`   pointer="true"/>`
			`</struct>`

bitfield[a]	bitfield attribute with the bitfield length	`struct BitfieldStruct {`
		`  short short_member: 1;`
		`  unsigned short`
		`  unsignedShort_member: 1;`
		`  short short_nmember_2: 0;`
		`  long long_member : 5;`
		`};`
			`<struct name="BitFieldStruct">`
			`  <member name="short_member"`
			`   type="short" bitField="1"/>`
			`  <member name="unsignedShort_member"`
			`   type="unsignedShort" bitField="1"/>`
			`  <member type="short" bitField="0"/>`
			`  <member name="long_member"`
			`   type="long" bitField="5"/>`
			`</struct>`

key directive [b]	key attribute with values true, false, 0 or 1
	
	Default (if not present): 0
		`struct KeyedPrimitiveStruct`
		`{`
		`  short short_member; //@key`
		`};`
			`<struct name="KeyedPrimitiveStruct">`
			`  <member name="short_member"`
			`   type="short" key="true"/>`
			`</struct>`

resolve-name directive [b]	resolveName attribute with values true, false, 0 or 1
	
	Default (if not present): 1
		`struct`
		`UnresolvedPrimitiveStruct {`
		`  PrimitiveStruct`
		`     primitive_member;`
		`     //@resolve-name false`
		`};`
			`<struct name=`
			`  "UnresolvedPrimitiveStruct">`
			`  <member name="primitive_member"`
			`   type="PrimitiveStruct"`
			`   resolveName="false"/>`
			`</struct>`

top-level directive [b]	topLevel attribute with values true, false, 0 or 1
	
	Default (if not present): 1
		`struct`
		`TopLevelPrimitiveStruct {`
		`    short short_member;`
		`}; //@top-level false`
			`<struct name="TopLevelPrimitiveStruct"`
			` topLevel="false">`
			`    <member name="short_member"`
			`     type="short"/>`
			`</struct>`

Other directives [b]	directive tag	`//@copy This text will be`
		`  copied in the generated`
		`  files`
			`<directive kind="copy">`
			`  This text will be copied in the`
			`  generated files`
			`</directive>`

Table 3.8 **Mapping Type System Constructs to XML**

Type/Construct		Example	
IDL	**XML**	**IDL**	**XML**

enum	enum tag	`enum PrimitiveEnum {`
		`ENUM1,`
		`ENUM2,`
		`ENUM3`
		`};`
			`<enum name="PrimitiveEnum">`
			`<enumerator name="ENUM1"/>`
			`<enumerator name="ENUM2"/>`
			`<enumerator name="ENUM3"/>`
			`</enum>`

		`enum PrimitiveEnum {`
		`ENUM1 = 10,`
		`ENUM2 = 20,`
		`ENUM3 = 30`
		`};`
			`<enum name="PrimitiveEnum">`
			`<enumerator name="ENUM1" value="10"/>`
			`<enumerator name="ENUM2" value="20"/>`
			`<enumerator name="ENUM3" value="30"/>`
			`</enum>`

constant	const tag	`const double PI = 3.1415;`	`<const name="PI" type="double"` `value="3.1415"/>`

struct	struct tag	`struct PrimitiveStruct {`
		`short short_member;`
		`};`
			`<struct name="PrimitiveStruct">`
			`<member name="short_member"`
			`type="short"/>`
			`</struct>`

union	union tag	`union PrimitiveUnion switch`
		`(long) {`
		`case 1:`
		`short short_member;`
		`case 2:`
		`case 3:`
		`float float_member;`
		`default:`
		`long long_member;`
		`};`
			`<union name="PrimitiveUnion">`
			`<discriminator type="long"/>`
			`<case>`
			`<caseDiscriminator value="1"/>`
			`<member name="short_member"`
			`type="short"/>`
			`</case>`
			`<case>`
			`<caseDiscriminator value="2"/>`
			`<caseDiscriminator value="3"/>`
			`<member name="float_member"`
			`type="float"/>`
			`</case>`
			`<case>`
			`<caseDiscriminator value="default"/>`
			`<member name="long_member"`
			`type="long"/>`
			`</case>`
			`</union>`

Table 3.8 **Mapping Type System Constructs to XML**

Type/Construct		Example	
IDL	**XML**	**IDL**	**XML**

valuetype	valuetype tag	valuetype BaseValueType {
		public long long_member;
		};
		
		valuetype DerivedValueType:
		BaseValueType {
		public long
		long_member_2;
		};
			<valuetype name="BaseValueType">
			<member name="long_member"
			type="long" visibility="public"/>
			</valuetype>
			
			<valuetype name="DerivedValueType"
			baseClass="BaseValueType">
			<member name="long_member_2"
			type="long" visibility="public"/>
			</valuetype>

typedef	typedef tag	typedef short ShortType;	<typedef name="ShortType" type="short"/>

		struct PrimitiveStruct {
		short short_member;
		};
		typedef PrimitiveStruct
		PrimitiveStructType;
			<struct name="PrimitiveStruct">
			<member name="short_member"
			type="short"/>
			</struct>
			
			<typedef name="PrimitiveStructType"
			type="nonBasic"
			nonBasicTypeName="PrimitiveStruct"/>

arrays	Attribute arrayDimensions	struct OneArrayStruct {
		short short_array[2];
		};
			<struct name="OneArrayStruct">
			<member name="short_array"
			type="short" arrayDimensions="2"/>
			</struct>

		struct TwoArrayStruct {
		short short_array[1][2];
		};
			<struct name="TwoArrayStruct">
			<member name="short_array"
			type="short" arrayDimensions="1,2"/>
			</struct>

bounded sequence	Attribute sequenceMaxLength > 0	struct SequenceStruct {
		sequence<short,4>
		short_sequence;
		};
			<struct name="SequenceStruct">
			<member name="short_sequence"
			type="short"
			sequenceMaxLength="4"/>
			</struct>

unbounded sequence	Attribute sequence-MaxLength set to -1	struct SequenceStruct {
		sequence<short>
		short_sequence;
		};
			<struct name="SequenceStruct">
			<member name="short_sequence"
			type="short" sequenceMaxLength="-1"/>
			</struct>

Table 3.8 **Mapping Type System Constructs to XML**

Type/Construct		Example	
IDL	**XML**	**IDL**	**XML**

array of sequences	Attributes sequenceMaxLength And arrayDimensions	struct
		ArrayOfSequencesStruct {
		  sequence<short,4>
		  short_sequence_array[2];
		};
			<struct name= "ArrayOfSequenceStruct">
			  <member name= "short_sequence_array"
			   type="short" arrayDimensions="2"
			   sequenceMaxLength="4"/>
			</struct>

sequence of arrays	Must be implemented with a typedef tag	typedef short
		    ShortArray[2];
		
		struct
		SequenceOfArraysStruct {
		  sequence<ShortArray,2>
		     short_array_sequence;
		};
			<typedef name="ShortArray"
			  type="short" dimensions="2"/>
			
			<struct name=
			  "SequenceOfArrayStruct">
			   <member name= "short_array_sequence"
			    type="nonBasic"
			    nonBasicTypeName="ShortSequence"
			    sequenceMaxLength="2"/>
			</struct>

sequence of sequences	Must be implemented with a typedef tag	typedef sequence<short,4>
		ShortSequence;
		
		struct
		SequenceOfSequencesStruct
		{
		  sequence<ShortSequence,2>
		  short_sequence_sequence;
		};
			<typedef name="ShortSequence"
			 type="short"sequenceMaxLength="4"/>
			<struct name="SequenceofSequencesStruct">
			  <member name="short_sequence_sequence"
			   type="nonBasic"
			   nonBasicTypeName="ShortSequence"
			   sequenceMax-Length="2"/>
			</struct>

module	module tag	module PackageName {
		  struct PrimitiveStruct {
		     long long_member;
		  };
		};
			<module name="PackageName">
			  <struct name="PrimitiveStruct">
			  <member name="long_member" type="long"/>
			  </struct>
			</module>

include	include tag	#include "PrimitiveTypes.idl"	<include file="PrimitiveTypes.xml"/>

a. Data types containing bitfield members are not supported by DynamicData (Section 3.8).

b. Directives are RTI extensions to the standard IDL grammar. For additional information about directives see Using Custom Directives (Section 3.3.8).

3.5 Creating User Data Types with XML Schemas (XSD)

You can describe data types with XML schemas (XSD), either independent or embedded in a Web Services Description Language (WSDL) file. The format is based on the standard IDL-to-WSDL mapping described in the OMG document "CORBA to WSDL/SOAP Interworking Specification." Defining a mapping between the DDS and WSDL type systems enables integration between Data Distribution Service and Web Services Technologies using WSDL.

Example Header for XSD:

```
<?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"/>

……

</xsd:schema>
```

Example Header for WSDL:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
 xmlns:dds="http://www.omg.org/dds"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
 xmlns:tns="http://www.omg.org/IDL-Mapped/"
 targetNamespace="http://www.omg.org/IDL-Mapped/">
 <types>
    <xsd:schema targetNamespace="http://www.omg.org/IDL-Mapped/">
      <xsd:import namespace="http://www.omg.org/dds"
       schemaLocation="rti_dds_topic_types_common.xsd"/>
……

    </xsd:schema>
  </types>
</definitions>
```

Table 3.9 describes how to map the DDS type system into XSD. The *RTI Data Distribution Service* code generator, *rtiddsgen*, will only accept XSD or WSDL files that follow this mapping.

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

char	dds:char[a]	struct PrimitiveStruct {
		    char char_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:element name="char_member"
			     minOccurs="1" maxOccurs="1"
			     type="dds:char">
			  </xsd:sequence>
			</xsd:complexType>

wchar	dds:wchar[a]	struct PrimitiveStruct {
		    wchar wchar_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:element name="wchar_member"
			     minOccurs="1" maxOccurs="1"
			     type="dds:wchar">
			  </xsd:sequence>
			</xsd:complexType>

octet	xsd:unsignedByte	struct PrimitiveStruct {
		    octet octet_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:element name="octet_member"
			     minOccurs="1" maxOccurs="1"
			     type="xsd:unsignedByte">
			  </xsd:sequence>
			</xsd:complexType>

short	xsd:short	struct PrimitiveStruct {
		    short short_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:element name="short_member"
			     minOccurs="1" maxOccurs="1"
			     type="xsd:short"/>
			  </xsd:sequence>
			</xsd:complexType>

unsigned short	xsd:unsignedShort	struct PrimitiveStruct {
		   unsigned short
		unsigned_short_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			  <xsd:element name="unsigned_short_member"
			     minOccurs="1" maxOccurs="1"
			     type="xsd:unsignedShort"/>
			  </xsd:sequence>
			</xsd:complexType>

long	xsd:int	struct PrimitiveStruct {
		    long long_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:element name="long_member"
			     minOccurs="1" maxOccurs="1"
			     type="xsd:int"/>
			  </xsd:sequence>
			</xsd:complexType>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

unsigned long	xsd:unsignedInt	struct PrimitiveStruct {
		    unsigned long
		unsigned_long_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:element name= "unsigned_long_member"
			      minOccurs="1" maxOccurs="1"
			      type="xsd:unsignedInt"/>
			  </xsd:sequence>
			</xsd:complexType>

long long	xsd:long	struct PrimitiveStruct {
		    long long
		long_long_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:elementname= "long_long_member"
			      minOccurs="1" maxOccurs="1"
			      type="xsd:long"/>
			  </xsd:sequence>
			</xsd:complexType>

unsigned long long	xsd:unsignedLong	struct PrimitiveStruct {
		    unsigned long long
		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	xsd: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	xsd: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>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

long double	dds:longDouble[a]	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	xsd: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	xsd: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>

bounded string	xsd:string with restriction to specify the maximum length	struct PrimitiveStruct {
		string<20> string_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			<xsd:sequence>
			<xsd:element name="string_member"
			minOccurs="1" maxOccurs="1">
			<xsd:simpleType>
			<xsd:restriction base="xsd:string">
			<xsd:maxLength value="20"
			fixed="true"/>
			</xsd:restriction>
			</xsd:simpleType>
			</xsd:element>
			</xsd:sequence>
			</xsd:complexType>

unbounded wstring	dds:wstring[a]	struct PrimitiveStruct {
		wstring wstring_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			<xsd:sequence>
			<xsd:element name="wstring_member"
			minOccurs="1" maxOccurs="1"
			type="dds:wstring"/>
			</xsd:sequence>
			</xsd:complexType>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

bounded wstring	xsd:wstring with restriction to specify the maximum length	`struct PrimitiveStruct {`
		`wstring<20>`
		`wstring_member;`
		`};`
			`<xsd:complexType name="PrimitiveStruct">`
			`<xsd:sequence>`
			`<xsd:element name="wstring_member"`
			`minOccurs="1" maxOccurs="1">`
			`<xsd:simpleType>`
			`<xsd:restriction base=`
			`"dds:wstring">`
			`<xsd:maxLength value="20"`
			`fixed="true"/>`
			`</xsd:restriction>`
			`</xsd:simpleType>`
			`</xsd:element>`
			`</xsd:sequence>`
			`</xsd:complexType>`

pointer	<!-- @pointer <true|false|1|0> -->
	Default (if not specified): false
		`struct PrimitiveStruct {`
		`long * long_member;`
		`};`
			`<xsd:complexType name="PrimitiveStruct">`
			`<xsd:sequence>`
			`<xsd:element name="long_member"`
			`minOccurs="1" maxOccurs="1"`
			`type="xsd:int"/>`
			`<!-- @pointer true -->`
			`</xsd:sequence>`
			`</xsd:complexType>`

bitfield[b]	<!-- @bitField <bitfield length> -->	`struct BitfieldStruct {`
		`short short_member: 1;`
		`unsigned short`
		`unsignedShort_member: 1;`
		`short: 0;`
		`long long_member: 5;`
		`};`
			`<xsd:complexType name="BitfieldStruct">`
			`<xsd:sequence>`
			`<xsd:element name="short_member"`
			`minOccurs="1" maxOccurs="1"`
			`type="xsd:short"/>`
			`<!-- @bitField 1 -->`
			`<xsd:element name='unsignedShort_member"`
			`minOccurs="1" maxOccurs="1"`
			`type="xsd:unsignedShort"/>`
			`<!-- @bitField 1 -->`
			`<xsd:element name="_ANONYMOUS_3"`
			`minOccurs="1" maxOccurs="1"`
			`type="xsd:short"/>`
			`<!-- @bitField 0 -->`
			`<xsd:element name="long_member"`
			`minOccurs="1" maxOccurs="1"`
			`type="xsd:int"/>`
			`<!-- @bitField 5 -->`
			`</xsd:sequence>`
			`</xsd:complexType>`

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

key directive[c]	<!-- @key
	<true|false|1|0>
	-->
	
	Default (if not specified): false
		struct
		KeyedPrimitiveStruct {
		long long_member; //@key
		};
			<xsd:complexType name="KeyedPrimitiveStruct">
			<xsd:sequence>
			<xsd:element name="long_member"
			minOccurs="1" maxOccurs="1"
			type="xsd:int"/>
			<!-- @key true -->
			</xsd:sequence>
			</xsd:complexType>

resolvename directive[c]	<!-- @resolveName
	<true|false|1|0>
	-->
	
	Default (if not specified): true
		struct
		UnresolvedPrimitiveStruct
		{
		PrimitiveStruct
		primitive_member;
		//@resolve-name false
		};
			<xsd:complexType name=
			"UnresolvedPrimitiveStruct">
			<xsd:sequence>
			<xsd:element name="primitive_member"
			minOccurs="1" maxOccurs="1"
			type="PrimitiveStruct"/>
			<!-- @resolveName false -->
			</xsd:sequence>
			</xsd:complexType>

top-level directive[c]	<!-- @topLevel
	<true|false|1|0> -->
	
	Default (if not specified): true
		struct
		TopLevelPrimitiveStruct {
		short short_member;
		}; //@top-level false
			<xsd:complexType
			name="TopLevelPrimitiveStruct">
			<xsd:sequence>
			<xsd:element name="short_member"
			minOccurs="1" maxOccurs="1"
			type="xsd:short"/>
			</xsd:sequence>
			</xsd:complexType>
			<!-- @topLevel false -->

other directives[c]	<!-- @<directive kind> <value> -->	//@copy This text will be copied in the generated files	<!--@copy This text will be copied in the generated files -->

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

enum	xsd:simpleType with enumeration	enum PrimitiveEnum {
		ENUM1,
		ENUM2,
		ENUM3
		};
		
		enum PrimitiveEnum {
		ENUM1 = 10,
		ENUM2 = 20,
		ENUM3 = 30
		};
			<xsd:simpleType name="PrimitiveEnum">
			<xsd:restriction base="xsd:string">
			<xsd:enumeration value="ENUM1"/>
			<xsd:enumeration value="ENUM2"/>
			<xsd:enumeration value="ENUM3"/>
			</xsd:restriction>
			</xsd:simpleType>
			
			<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>

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

struct	xsd:complexType with xsd:sequence	struct PrimitiveStruct {
		short short_member;
		};
			<xsd:complexType name="PrimitiveStruct">
			<xsd:sequence>
			<xsd:element name="short_member"
			minOccurs="1" maxOccurs="1"
			type="xsd:short"/>
			</xsd:sequence>
			</xsd:complexType>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

union	xsd:complexType with xsd:choice	union PrimitiveUnion
		switch (long) {
		case 1:
		short short_member;
		default:
		long long_member;
		};
			<xsd:complexType name="PrimitiveUnion">
			<xsd:sequence>
			<xsd:element name="discriminator"
			type="xsd:int"/>
			<xsd:choice>
			<!-- case 1 -->[d]
			<xsd:element name="short_member"
			minOccurs="0" maxOccurs="1"
			type="xsd:short">
			<xsd:annotation>
			<xsd:appinfo>
			<case>1</case>
			</xsd:appinfo>
			</xsd:annotation>
			</xsd:element>
			<!-- case default -->
			<xsd:element name="long_member"
			minOccurs="0" maxOccurs="1"
			type="xsd:int">
			<xsd:annotation>
			<xsd:appinfo>
			<case>default</case>
			</xsd:appinfo>
			</xsd:annotation>
			</xsd:element>
			</xsd:choice>
			</xsd:sequence>
			</xsd:complexType>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

valuetype	xsd:complexType with @valuetype directive	`valuetype BaseValueType {`
		`  public long`
		`long_member;`
		`};`
		
		`valuetype`
		`DerivedValueType:`
		`BaseValueType {`
		`  public long`
		`     long_member2;`
		`  public long`
		`     long_member3;`
		`};`
			`<xsd:complexType name="BaseValueType">`
			`  <xsd:sequence>`
			`     <xsd:element name=”long_member"`
			`      maxOccurs="1" minOccurs="1"`
			`      type="xs:int"/>`
			`      <!-- @visibility public -->`
			`  </xsd:sequence>`
			`</xs:complexType>`
			`<!-- @valuetype true -->`
			
			`<xs:complexType name="DerivedValueType">`
			`  <xs:complexContent>`
			`     <xs:extension base="BaseValueType">`
			`         <xs:sequence>`
			`            <xs:element name= "long_member2"`
			`             maxOccurs="1" minOccurs="1"`
			`             type="xs:int"/>`
			`            <!-- @visibility public -->`
			`            <xs:element name= "long_member3"`
			`             maxOccurs="1" minOccurs="1"`
			`             type="xs:int"/>`
			`            <!-- @visibility public -->`
			`         </xs:sequence>`
			`     </xs:extension>`
			`  </xs:complexContent>`
			`</xs:complexType>`
			`<!-- @valuetype true -->`

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

typedef	Type definitions are mapped to XML schema type restrictions	`typedef short ShortType;`
		
		`struct PrimitiveStruct {`
		`  short short_member;`
		`};`
		
		`typedef PrimitiveType`
		`PrimitiveStructType;`
			<xsd:simpleType name="ShortType">
			  <xsd:restriction base="xsd:short"/>
			</xsd:simpleType>
			
			<!-- Struct definition -->
			<xsd:complexType name="PrimitiveStruct">
			  <xsd:sequence>
			    <xsd:element name="short_member"
			     minOccurs="1" maxOccurs="1"
			     type="xsd:short"/>
			  </xsd:sequence>
			</xsd:complexType>
			
			<!-- Typedef definition -->
			<xsd:complexType
			  name="PrimitiveTypeStructType">
			  <xsd:complexContent>
			    <xsd:restriction base=”PrimitiveStruct”>
			      <xsd:sequence>
			        <xsd:element name="short_member"
			         minOccurs="1" maxOccurs="1"
			         type="xsd:short"/>
			      </xsd:sequence>
			    </xsd:restriction>
			  </xsd:complexContent>
			</xsd:complexType>

Table 3.9 **Mapping Type System Constructs to XSD**

<table>
<tr><th colspan="2">Type/Construct</th><th colspan="2">Example</th></tr>
<tr><th>IDL</th><th>XSD</th><th>IDL</th><th>XSD</th></tr>
<tr><td>arrays</td><td>n xsd:complexType with sequence containing one element with min & max occurs

There is one xsd:complexType per array dimension</td><td><pre>struct OneArrayStruct {
  short short_array[2];
};</pre></td><td><pre><!-- 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></pre></td></tr>
</table>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

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_ArrayOfShort">
			<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_ArrayOfArrayOfShort">
			<xsd:sequence>
			<xsd:element name="item"
			minOccurs="1" maxOccurs="1"
			type=
			"TwoArrayStruct_short_array_ArrayOfShort">
			</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_ArrayOfArrayOfShort"/>
			</xsd:sequence>
			</xsd:complexType>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

bounded sequence	xsd:complexType with sequence containing one element with min & max occurs	struct SequenceStruct {
		sequence<short,4>
		short_sequence;
		};
			<!-- Sequence type -->
			<xsd:complexType name=
			"SequenceStruct_short_sequence_SequenceOfShort">
			<xsd:sequence>
			<xsd:element name="item" minOccurs="0"
			maxOccurs="4" type="xsd:short">
			</xsd:element>
			</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>

unbound-ed sequence	xsd:complexType with sequence containing one element with min & max occurs	struct SequenceStruct {
		sequence<short>
		short_sequence;
		};
			<!-- 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 an 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 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

array of sequences	n + 1 xsd:complexType with sequence containing one element with min & max occurs
	
	There is one xsd:complexType per array dimension and one xsd:complexType for the sequence
		`struct ArrayOfSequencesStruct {`
		`  sequence<short,4>`
		`  sequence_sequence[2];`
		`};`
			`<!-- Sequence declaration -->`
			`<xsd:complexType`
			`  name=`
			`"ArrayOfSequencesStruct_sequence_array_SequenceOf`
			`Short">`
			`  <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_array_ArrayOf`
			`SequenceOfShort">`
			`  <xsd:sequence>`
			`    <xsd:element name="item"`
			`     minOccurs="2" maxOccurs="2"`
			`     type=`
			`"ArrayOfSequencesStruct_sequence_array_SequenceOf`
			`Short">`
			`    </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_array_ArrayOf`
			`SequenceOfShort"/>`
			`    </xsd:sequence>`
			`</xsd:complexType>`

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

sequence of arrays	Sequences of arrays must be implemented using an explicit type definition (typedef) for the array	typedef short
		ShortArray[2];
		
		struct
		SequenceOfArraysStruct {
		sequence<ShortArray,2>
		arrays_sequence;
		};
			<!-- Array declaration -->
			<xsd:complexType name="ShortArray">
			<xsd:sequence>
			<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_SequenceO
			fShortArray">
			<xsd:sequence>
			<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:sequence>
			<xsd:element name="arrays_sequence"
			minOccurs="1" maxOccurs="1"
			type=
			"SequencesOfArraysStruct_arrays_sequence_Sequence
			OfShortArray"/>
			</xsd:sequence>
			</xsd:complexType>

Table 3.9 **Mapping Type System Constructs to XSD**

Type/Construct		Example	
IDL	**XSD**	**IDL**	**XSD**

sequence of sequences	Sequences of sequences must be implemented using an explicit type definition (typedef) for the second sequence	typedef sequence<short,4>
		ShortSequence;
		
		struct
		SequenceOfSequences {
		sequence<ShortSequence, 2>
		sequences_sequence;
		};
			<!-- 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>
			
			<!-- External sequence declaration -->
			<xsd:complexType name=
			"SequencesOfSequences_sequences_sequence_Sequence
			OfShortSequence">
			<xsd:sequence>
			<xsd:element name="item"
			minOccurs="0" maxOccurs="2"
			type="ShortSequence">
			</xsd:element>
			</xsd:sequence>
			</xsd:complexType>
			
			<!--Struct containing a sequence of sequences -->
			<xsd:complexType name="SequenceOfSequences">
			<xsd:sequence>
			<xsd:element name="sequences_sequence"
			minOccurs="1" maxOccurs="1"
			type="SequencesOfSequences_
			sequences_sequence_SequenceOfShortSequence"/>
			</xsd:sequence>
			</xsd:complexType>

module	Modules are mapped adding the name of the module before the name of each type inside the module	module PackageName {
		struct PrimitiveStruct {
		long long_member;
		};
		};
			<xsd:complexType name=
			"PackageName.PrimitiveStruct">
			<xsd:sequence>
			<xsd:element name="long_member"
			minOccurs="1" maxOccurs="1"
			type="xsd:int"/>
			</xsd:sequence>
			</xsd:complexType>

include	xsd:include	#include "PrimitiveType.idl"	<xsd:include schemaLocation= "PrimitiveType.xsd"/>

a. All files that use the primitive types char, wchar, long double and wstring must reference rti_dds_topic_types_common.xsd. See Primitive Types (Section 3.5.1).

b. Data types containing bitfield members are not supported by DynamicData (Section 3.8).

c. Directives are RTI extensions to the standard IDL grammar. For additional information about directives see Using Custom Directives (Section 3.3.8).

d. 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.

3.5.1 Primitive Types

The primitive types char, wchar, long double, and wstring are not supported natively in XSD. *RTI Data Distribution Service* provides definitions for these types in the file **<NDDSHOME>/resource/qos_profiles_4.5x[1]/schema/rti_dds_topic_types_common.xsd**. All files that use the primitive types char, wchar, long double and wstring must reference rti_dds_topic_types_common.xsd. For example:

```
<?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 rtiddsgen

The *rtiddsgen* utility provided by *RTI Data Distribution Service* creates the code needed to define and register a user data type with *RTI Data Distribution Service*. Using this tool is optional if:

❏ You are using dynamic types (see Managing Memory for Built-in Types (Section 3.2.8))

1. Replace the `x` in `4.5x` with the version letter for the current *RTI Data Distribution Service* release.

❑ You are using one of the built-in types (see Built-in Data Types (Section 3.2))

To use *rtiddsgen*, you must supply a description of the type in an IDL, XML, XSD, or WSDL file. The supported syntax for each one of the notations is described in Section 3.8.5.1 (IDL), Section 3.4 (XML) and Section 3.5 (XSD and WSDL). You can define multiple data types in the same type-definition file.

Table 3.10 on page 3-108 (for C, C++, and C++/CLI and C#) and Table 3.11 on page 3-109 (for Java) show the files that *rtiddsgen* creates for an example IDL file called **Hello.idl**. (The file extension will depend on the chosen language: **.c** for C, **.cxx** for C++, **.cpp** for C++/CLI, **.cs** for C#.)

On Windows systems: Before running *rtiddsgen*, run **VCVARS32.BAT** in the same command prompt that you will use to run *rtiddsgen*.

Table 3.10 **Files Created by rtiddsgen for C, C++, C++/CLI, C# for Example "Hello.idl"**

Generated Files	Description
Required files for the user data type. The source files should be compiled and linked with the user application. The header files are required to use the data type in source. *You should not modify these files unless you intend to customize the generated code supporting your type.*	
Hello.[c,cxx, cpp] HelloSupport.[c, cxx, cpp] HelloPlugin.[c,cxx, cpp]	Generated code for the data types. These files contain the implementation for your data types.
Hello.h HelloSupport.h HelloPlugin.h	Header files that contain declarations used in the implementation of your data types.
Optional files generated when you use the "-example <arch>" command-line option. *You may modify and use these files as a way to create simple applications that publish or subscribe to the user data type.*	
Hello_publisher.[c, cxx, cpp, cs]	Example code for an application that publishes the user data type. This example shows the basic steps to create all of the *RTI Data Distribution Service* objects needed to send data. You will need to modify the code to set and change the values being sent in the data structure. Otherwise, just compile and run.
Hello_subscriber.[c, cxx, cpp,cs]	Example code for an application that subscribes to the user data type. This example shows the basic steps to create all of the *RTI Data Distribution Service* objects needed to receive data using a "listener" function. No modification of this file is required. It is ready for you to compile and run.

Table 3.10 **Files Created by rtiddsgen for C, C++, C++/CLI, C# for Example "Hello.idl"**

Generated Files	Description
Hello.dsw or Hello.sln, Hello_publisher.dsp or Hello_publisher.vcproj, Hello_subscriber.dsp or Hello_subscriber.vcproj	Microsoft Visual C++ or Visual Studio .NET Project workspace and project files, generated only for "i86Win32" architectures. To compile the generated source code, open the workspace file and build the two projects.
makefile_Hello_<architecture>	Makefile for non-Windows-based architectures. An example <architecture> would be **linux2.4gcc3.2.2**.

Table 3.11 **Files Created by rtiddsgen for Java for Example "Hello.idl"**

Data Type	Generated Files	Description
Since the Java language requires individual files to be created for each class, rtiddsgen *will generate a source file for every IDL construct that translates into a class in Java.*		
Constants	<Name>.java	Class associated with the constant
Enums	<Name>.java	Class associated with enum type
Structures/ Unions	<Name>.java <Name>Seq.java <Name>DataReader.java <Name>DataWriter.java <Name>TypeSupport.java	Structure/Union class Sequence class *RTI Data Distribution Service DataReader* and *DataWriter* classes Support (serialize, deserialize, etc.) class
Typedef of sequences or arrays	<Name>.java <Name>Seq.java <Name>TypeSupport.java	Wrapper class Sequence class Support (serialize, deserialize, etc.) class

Table 3.11 **Files Created by rtiddsgen for Java for Example “Hello.idl”**

Data Type	Generated Files	Description
Optional files generated when you use the “-example <arch>” command-line option. You may modify and use these files as a way to create simple applications that publish or subscribe to the user data type.		
Structures/ Unions	<Name>Publisher.java <Name>Subscriber.java	Example code for applications that publish or subscribe to the user data type. You should modify the code in the publisher application to set and change the value of the published data. Otherwise, both files should be ready to compile and run.
	makefile_Hello_<architecture>	Makefile for non-Windows-based architectures. An example <architecture> is linux2.4gcc3.2.2.
Structures/ Unions/ Typedefs/ Enums	<Name>TypeCode.java (Note: this is *not* generated if you use -notypecode)	Type code class associated with the IDL type given by <Name>.

NOTE: Before using an *rtiddsgen*-generated makefile to compile an application, make sure the **${NDDSHOME}** environment variable is set as described in the *Getting Started Guide*. For INTEGRITY architectures, **${NDDSHOME}** must be set when generating the project files.

3.6.1 rtiddsgen Command-Line Arguments

There are several command-line options you can pass to *rtiddsgen*:

Note: CORBA support requires the RTI CORBA Compatibility Kit

```
rtiddsgen [-d <outdir>]
          [-language <C|C++|Java|C++/CLI|C#>]
          [-namespace] (C++ only)
          [-package <packagePrefix>] (Java only)
          [-example <arch>]
          [-replace]
          [-debug]
          [-corba [client header file]] [-orb \<CORBA ORB\>]]
          [-optimization <level of optimization>]
          [-stringSize <Unbounded strings size>]
          [-sequenceSize <Unbounded sequences size>]
          [-notypecode]
          [-ppDisable]
          [-ppPath <preprocessor executable>]
          [-ppOption <option>]
          [-D <name>[=<value>]]
          [-U <name>]
          [-I <directory>]
          [-noCopyable]
          [-use42eAlignment]
          [-enableEscapeChar]
          [-typeSequenceSuffix <Suffix>]
          [-convertToXml  |
           -convertToXsd  |
           -convertToWsdl |
           -convertToIdl]
          [-convertToCcl]
          [-convertToCcs]
          [-version]
          [-help]
          [-verbosity [1-3]]
          [[-inputIdl] <IDLInputFile.idl> |
           [-inputXml] <XMLInputFile.xml> |
           [-inputXsd] <XSDInputFile.xsd> |
           [-inputWsdl] <WSDLInputFile.wsdl>]
```

Table 3.12 describes the options (in alphabetical order).

Table 3.12

Option	Description
-convertToCcl	Converts the input type description file into CCL format. This option creates a new file with the same name as the input file and a .ccl extension.
-convertToCcs	Converts the input type description file into CCs format. This option creates a new file with the same name as the input file and a .ccs extension.
-convertToIdl	Converts the input type description file into IDL format. This option creates a new file with the same name as the input file and a .idl extension.
-convertToWsdl	Converts the input type description file into WSDL format. This option creates a new file with the same name as the input file and a .wsdl extension.
-convertToXml	Converts the input type description file into XML format. This option creates a new file with the same name as the input file and a .xml extension.
-convertToXsd	Converts the input type description file into XSD format. This option creates a new file with the same name as the input file and a .xsd extension.
-corba	This option is only available when using the *RTI CORBA Compatibility Kit for RTI Data Distribution Service* (available for purchase as a separate product). Please see Part 6: RTI CORBA Compatibility Kit.
-D <name>[=<value>]	Defines preprocessor macros.
-d	Generates the output in the specified directory. By default, *rtiddsgen* will generate files in the directory where the input type-definition file is found.
-debug	Creates XML files for debugging *rtiddsgen* only. Use this option only at the direction of RTI support; it is unlikely to be useful to you otherwise.
-enableEscapeChar	Enables use of the escape character '_' in IDL identifiers. When **-corba** is used, this option is always enabled.
-example <arch>	Generates example application code and makefiles (for UNIX-based systems) or workspace and project files (for Windows systems) based on the type-definition file. The parameter specifies the architecture for the example makefiles. Valid options for **<arch>** are listed in the Platform Notes.
IDLInputFile.idl	File containing IDL descriptions of your data types. If **-inputIdl** is not used, the file must have an .idl extension.
-I <directory>	Adds to the list of directories to be searched for type-definition files (IDL, XML, XSD or WSDL files). Note: A type-definition file in one format cannot include a file in another format.
-inputIdl	Indicates that the input file is an IDL file, regardless of the file extension.
-inputWsdl	Indicates that the input file is a WSDL file, regardless of the file extension.
-inputXml	Indicates that the input file is a XML file, regardless of the file extension.

Table 3.12

Option	Description
`-inputXsd`	Indicates that the input file is a XSD file, regardless of the file extension.
`-help`	Prints out the command line options for *rtiddsgen*.
`-language`	Specifies the language to use for the generated files. The default language is C++; you can also choose C, C++/CLI, C#, or Java.
`-namespace`	Specifies the use of C++ namespace. (For C++ only. For C++/CLI and C#, it is implied-namespaces are always used.)
`-noCopyable`	Forces *rtiddsgen* to put 'copy' logic into the corresponding TypeSupport class rather than the type itself. This option is only used for Java code generation. This option is not compatible with the use of **ndds_standalone_type.jar** (see Section 3.7). Note that when generating code for Java, the **-corba** option implies the **-noCopyable** option (whether or not you specify **-noCopyable**).[a]
`-notypecode`	Disables type-code support. By using this option, you can generate code that can be used in a standalone manner—see Using Generated Types without RTI Data Distribution Service (Standalone) (Section 3.7). Note: If you are using a large data type (more than 64 K) *and* type code support, you will see a warning when type code information is sent. *RTI Data Distribution Service* has a type code size limit of 64K. To avoid the warning when working with data types with type codes larger than 64K, turn off type code support by using **-notypecode**.
`-replace`	Allows *rtiddsgen* to overwrite any existing generated files. If it is not present and existing files are found, *rtiddsgen* will print a warning but will not overwrite them.
`-optimization`	See Optimizing Typedefs (-optimization) (Section 3.6.1.1 on Page 3-115)
`-orb`	Specifies the CORBA ORB. The majority of code generated is independent of the ORB. However, for some IDL features the code generated depends on the ORB. *rtiddsgen* generates code compatible with ACE-TAO or JacORB. To select an ACE_TAO version use the -orb parameter. The default is ACE_TAO1.6. This option can only be used with the **-corba** option.
`-package`	Specifies the root package into which generated classes will be placed. It applies to Java only. If the type-definition file contains module declarations, those modules will be considered subpackages of the package specified here.
`-ppDisable`	Disables the preprocessor.
`-ppOption <option>`	Specifies a preprocessor option. This parameter can be used multiple times to provide the command-line options for the specified preprocessor. See **-ppPath**.

3. User Data Types

Table 3.12

Option	Description

-ppPath <preprocessor executable>	Specifies the preprocessor. If you only specify the name of an executable (not a complete path to that executable), the executable must be found in your **Path**. The default value is "**cpp**" for non-Windows architectures and "**cl.exe**" for Windows architectures.If you use **-ppPath** to provide the full path and filename for **cl.exe** or the **cpp** preprocessor, you must also use **-ppOption** (described below) to set the following preprocessor options:
	If you use a non-default path for cl.exe, you also need to set:
	`-ppOption /nologo -ppOption /C -ppOption /E -ppOption /X`
	If you use a non-default path for cpp, you also need to set:
	`-ppOption -C`

-sequenceSize	Sets the size assigned to unbounded sequences. The default value is 100 elements.
-stringSize	Sets the size assigned to unbounded strings, not counting a terminating NULL character. The default value is 255 bytes.
-typeSequenceSuffix <suffix>	This option assigns a suffix to the names of the implicit sequences defined for IDL types. It only applies if **-corba** is also specified. By default, the suffix is 'Seq'. Therefore, given the type 'Foo' the name of the implicit sequence will be 'FooSeq'.
-U <name>	Cancels any previous definition of <name>.
-use42eAlignment	Makes the generated code compatible with *RTI Data Distribution Service* 4.2e. This option should be used when compatibility with 4.2e is required and the topic data types contain double, long long, unsigned long long, or long double members.
-verbosity [1-3]	*rtiddsgen* verbosity: 1: exceptions 2: exceptions and warnings 3: exceptions, warnings and information (Default)
-version	Displays the version of *rtiddsgen* being used.
WSDLInputFile.wsdl	WSDL file containing XSD descriptions of your data types. If **-inputWsdl** is not used, the file must have an .wsdl extension.
XMLInputFile.idl	File containing XML descriptions of your data types. If **-inputXml** is not used, the file must have an .xml extension.
XSDInputFile.xsd	File containing XSD descriptions of your data types. If **-inputXsd** is not used, the file must have an .xsd extension.

a. CORBA support is only available when using the *RTI CORBA Compatibility Kit for RTI Data Distribution Service* (available for purchase as a separate product). See Part 6: RTI CORBA Compatibility Kit.

3.6.1.1 Optimizing Typedefs (-optimization)

The **-optimization** option specifies how support for **typedefs** is generated in C and C++ code. This option is only useful when there are **typedefs** defined in the IDL file. This option only applies to C and C++ because the Java language does not contain the typedef construct. In other words, *rtiddsgen* always resolves typedef'ed names to their most basic types when generating Java code (except for typedefs of arrays and sequences which are converted to wrapper classes—see Note 8 on Page 3-74). Effectively, Java code is always generated with an equivalent optimization level of 2. Choices are:

- **0** (default): No optimization. Typedef'ed types are treated as full types and type-plugin and support code is generated and invoked when the typedefs are used in other structures.
- **1**: The compiler generates type-plugin and support code for typedefs but optimizes its use. If a type is a typedef that can be resolved either to a primitive type (char, short, long, etc.) or to another type that is defined in the same IDL file, then when the typedef is used in the definition of another structure, *rtiddsgen* will generate code that invokes the plugin and support code of the most basic type to which the typedef can be resolved.

  This will save at least one function call for serialization, deserialization, and other manipulation of the parent structure. This optimization level is always safe to use unless the user intends to modify the generated type-plugin and support code. In that case, we recommend using the default of no optimization of typedefs.
- **2**: Same as level **1** with the addition that the type-plugin and support code for typedefs are not generated (since they would not be used by the code for data types defined in the same IDL file that uses the typedefs).

  This typedef optimization level is only recommend if you only have a *single* IDL file that contains the definitions of all of the user data types passed by *RTI Data Distribution Service* on the network. If you have multiple IDL files, and types defined in one file use typedefs that are defined in another, then *rtiddsgen* will generate code assuming that the type-plugin and support code were generated for the typedef'ed types. If level 2 optimization was used when generating the code for the IDL file that contained the typedefs, then the plugin and support code for the typedefs would not have been generated, and compilation and linking errors will result.

For example, consider this declaration:

```
typedef short MyShort

struct MyStructure {
    MyShort member;
};
```

With optimization **0**: The type-plugin and support code for **MyStructure** will use the generated code for **MyShort** to serialize, deserialize, or otherwise manipulate the **member** field of **MyStructure**.

With optimization **1**: The type plug-in and support code for **MyStructure** will directly serialize, deserialize or otherwise manipulate the **member** field of **MyStructure** as a **short—**saving a function call. However, plugin and support code for **MyShort** is still generated because it would be used by the code generated from another IDL file that refers to **MyShort**.

With optimization **2**: The type-plugin and support code for **MyStructure** will directly serialize, deserialize or otherwise manipulate the **member** field of **MyStructure** as a **short**–saving a function call. In addition, no plugin or support code for **MyShort** is generated.

3.7 Using Generated Types without RTI Data Distribution Service (Standalone)

You can use the generated type-specific source and header files without linking the *RTI Data Distribution Service* libraries or even including the *RTI Data Distribution Service* header files. That is, the generated files for your data types can be used standalone.

The directory **<NDDSHOME>/resource/rtiddsgen/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.

Note: You must use *rtiddsgen's* **-notypecode** option to generate code for standalone use.

3.7.1 Using Standalone Types in C

The generated files that can be used standalone are:

- **<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 rtiddsgen-generated types in a standalone manner:

1. Make sure you use *rtiddsgen's* **-notypecode** option to generate the code.
2. Include the directory **<NDDSHOME>/resource/rtiddsgen/standalone/include** in the list of directories to be searched for header files.
3. Add the source files, **ndds_standalone_type.c** and **<idl file name>.c**, to your project.
4. Include the file **<idl file name>.h** in the source files that will use the generated types in a standalone manner.
5. Compile the project using the following two preprocessor definitions:
   a. **NDDS_STANDALONE_TYPE**
   b. 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++

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. Make sure you use *rtiddsgen's* **-notypecode** option to generate the code.
2. Include the directory **<NDDSHOME>/resource/rtiddsgen/standalone/include** in the list of directories to be searched for header files.

3. Add the source files, **ndds_standalone_type.cxx** and **<idl file name>.cxx,** to your project.
4. Include the file **<idl file name>.h** in the source files that will use the *rtiddsgen* types in a standalone manner.
5. Compile the project using the following two preprocessor definitions:
   a. **NDDS_STANDALONE_TYPE**
   b. 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 type>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. Make sure you use *rtiddsgen's* **-notypecode** option to generate the code.
2. Include the file **ndds_standalone_type.jar** in the classpath of your project.
3. 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 Introduction to TypeCode

Type schemas—the names and definitions of a type and its fields—are represented by TypeCode objects. 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.

```
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_LONGLONG,
  TK_ULONGLONG,
  TK_LONGDOUBLE,
  TK_WCHAR,
  TK_WSTRING,
  TK_VALUE,
  TK_SPARSE
}
```

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 online documentation (select **Modules**, **Topic Module, Type Code Support**).

Type codes are enabled by default when you run *rtiddsgen*. The **-notypecode** option disables generation of type code information. Type-code support does increase the amount of memory used, so if you need to save on memory, you may consider disabling type codes. See rtiddsgen Command-Line Arguments (Section 3.6.1).

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

3. User Data Types

3.8.2 Defining New Types

Locally, your application can access the type code for a generated type "Foo" by calling the **Foo_get_typecode()** operation in the code for the type generated by *rtiddsgen* (unless type-code support is disabled with the **-notypecode** option). 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 {
    long my_integer;
    float my_float;
    bool my_bool;
    string<128> my_string; // @key
};
```

This is how you would define the same type at run time in C++:

```
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_MEMBER, ex);

structTc->add_member("my_float", DDS_TYPECODE_MEMBER_ID_INVALID,
                     factory->get_primitive_tc(DDS_TK_FLOAT),
                     DDS_TYPECODE_NONKEY_MEMBER, ex);

structTc->add_member("my_bool", DDS_TYPECODE_MEMBER_ID_INVALID,
                     factory->get_primitive_tc(DDS_TK_BOOLEAN),
                     DDS_TYPECODE_NONKEY_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 *RTI Data Distribution Service* online documentation, which is available in HTML and PDF formats 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:

```
DDS_StructMemberSeq structMembers;
structMembers.ensure_length(4, 4);
DDS_TypeCodeFactory* factory = DDS_TypeCodeFactory::get_instance();

structMembers[0].name = DDS_String_dup("my_integer");
structMembers[0].type = factory->get_primitive_tc(DDS_TK_LONG);

structMembers[1].name = DDS_String_dup("my_float");
structMembers[1].type = factory->get_primitive_tc(DDS_TK_FLOAT);

structMembers[2].name = DDS_String_dup("my_bool");
structMembers[2].type = factory->get_primitive_tc(DDS_TK_BOOLEAN);

structMembers[3].name = DDS_String_dup("my_string");
structMembers[3].type = factory->create_string_tc(128);
structMembers[3].is_key = DDS_BOOLEAN_TRUE;

DDS_ExceptionCode_t ex = DDS_NO_EXCEPTION_CODE;
DDS_TypeCode* structTc = factory->create_struct_tc(
                                        "MyType", structMembers, ex);
```

After you have defined the TypeCode, you will register it with a *DomainParticipant* using a logical name. You will use this logical name later when you create a *Topic*.

```
DDSDynamicDataTypeSupport* type_support =
    new DDSDynamicDataTypeSupport(
              structTc,   DDS_DYNAMIC_DATA_TYPE_PROPERTY_DEFAULT);

DDS_ReturnCode_t retcode = type_support->register_type(participant,
                        "My Logical Type Name");
```

Now that you have created a type, you will need to know how to interact with objects of that type. Continue reading Section 3.8.3 below for more information.

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 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, *RTI Data Distribution Service* supports sparse value types. A sample of such a type only contains the field values that were explicitly set by the sender. A recipient of that sample will receive an error when trying to look up the value of any other field.

An endpoint (*DataWriter* or *DataReader*) using a sparse value type will not communicate with another endpoint using a non-sparse value type or structure type, even if the two types contain similar member definitions, because these kinds of types have different semantics. A structure or non-sparse value type is a commitment to provide exactly the data described by the type's members and in a certain order. In contrast, a sparse value type is a commitment to provide some subset of those data values in no particular order.

Because direct programming language representations of data types typically have no way to express the concept of sparse fields (there is no way, for example, for a C structure to omit some of its fields), using sparse types requires use of the dynamic type API described in Defining New Types (Section 3.8.2). You will use the Dynamic Data API to work with sparse samples, just as you would with samples of any other dynamically defined type. For more information about working with sparse samples, see Objects of Dynamically Defined Types (Section 3.9.2) or the online (HTML) documentation.

A sparse version of the "MyType" type described above would be defined like this:

```
DDS_ExceptionCode_t ex = DDS_NO_EXCEPTION_CODE;
DDS_TypeCodeFactory* factory = DDS_TypeCodeFactory::get_instance();
DDS_TypeCode* sparseTc = factory->create_sparse_tc(
                          "MySparseType", DDS_VM_NONE, NULL, ex);

// add members
sparseTc->add_member("my_integer", ID_MY_INTEGER,
                      factory->get_primitive_tc(DDS_TK_LONG),
                      DDS_TYPECODE_NONKEY_MEMBER, ex);
```

```
sparseTc->add_member("my_float", ID_MY_FLOAT,
                        factory->get_primitive_tc(DDS_TK_FLOAT),
                        DDS_TYPECODE_NONKEY_MEMBER, ex);
sparseTc->add_member("my_bool", ID_MY_BOOL,
                        factory->get_primitive_tc(DDS_TK_BOOLEAN),
                        DDS_TYPECODE_NONKEY_MEMBER, ex);
sparseTc->add_member("my_string", ID_MY_STRING,
                        factory->create_string_tc(128),
                        DDS_TYPECODE_KEY_MEMBER, ex);
```

Detailed descriptions of the methods and constants you see above can be found in the *RTI Data Distribution Service* API documentation.

Integral to the definition of a sparse type are the *member IDs* of its fields. An ID is a two-byte integer that uniquely identifies a field within its parent type; these IDs are chosen by the type's designer. (In the code example above, ID_MY_INTEGER, ID_MY_FLOAT, and ID_MY_BOOL are examples of user-defined symbolic constants representing member ID values.) When a sparse sample is serialized, the middleware will embed the IDs of the fields that are present, so that recipients will know how to deserialize it.

Although member IDs are a relatively efficient way to describe a sample's contents, they do use network bandwidth. This can be an important issue if you are considering using sparse types to decrease the size of your data samples on the network. Although the relative cost of adding member IDs to your packets will vary depending on the sizes and layout of your fields, *the following is a good rule of thumb*: if you expect a given data sample to contain less than half of the fields that are legal for its type, sparse types will probably save you on bandwidth. If, on the other hand, most samples contain most fields, you will probably be better off using a plain structure type and simply ignoring irrelevant fields on the receiving side.

3.8.4 Type Extension and Versioning

As your system evolves, you may find that your data types need to change. And unless your system is relatively small, you may not be able to bring it all down at once in order to modify them. Instead, you may need to upgrade your types one component at a time—or even on the fly, without bringing any part of the system down.

You can use the sparse types described above to efficiently version types—and not just at the level of entire types, but at the level of individual fields.

- ❑ **You can add new fields to a type at any time.** Because the type is sparse, existing publishers of the type that have not been updated will simply omit the new field in any data samples they send. If you anticipate changing your types in future versions of your system, make sure that you ignore fields that you do not recognize, so that your application will be robust to future type changes.

```
sparseTc->add_member("myNewInteger", ID_MY_NEW_INTEGER,
                DDS_TheTypeCodeFactory->get_primitive_tc(TK_LONG),
                DDS_TYPECODE_NONKEY_MEMBER, ex);
```

- ❑ **You cannot remove fields from an existing type.** Doing so would break older applications and invalidate historical samples that might already be in the caches of upgraded applications. Instead, simply stop sending values for the fields you wish to deprecate.

3.8.5 Sending TypeCodes on the Network

In addition to being used locally, serialized type codes are typically published automatically during discovery as part of the built-in topics for publications and subscriptions. See Built-in DataReaders (Section 14.2). This allows applications to publish or subscribe to topics of arbitrary types. This functionality is useful for generic system monitoring tools like the *rtiddsspy* debug tool (in the online documentation, select **Modules**, **Programming Tools**).

Note: Type codes are not cached by *RTI Data Distribution Service* upon receipt and are therefore not available from the built-in data returned by the *DataWriter's* **get_matched_subscription_data()** operation or the *DataReader's* **get_matched_publication_data()** operation.

If your data type has an especially complex type code, you may need to increase the value of the **type_code_max_serialized_length** field in the *DomainParticipant's* DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4). Or, to prevent the propagation of type codes altogether, you can set this value 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 TypeCode propagation.

3.8.5.1 Type Codes for Built-in Types

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

```
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 3.13.

3.9 Working with 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 Built-in Data Types (Section 3.2)), dynamically defined types (see Managing Memory for Built-in Types (Section 3.2.8)), or code generated from IDL, XML, XSD, or WSDL files (see Sections 3.3 through 3.5).

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 *RTI Data Distribution Service* data type is like any other data type in your

Table 3.13 **Properties for Allocating Size of Built-in Types, per DomainParticipant**

Built-in Type	Property	Description
String	dds.builtin_type.string.max_size	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
Keyed-String	dds.builtin_type.keyed_string.max_key_size	Maximum size of the keys used by the *DataWriters* and *DataReaders* belonging to a *DomainParticipant* (includes the NULL-terminated character). Default: 1024
	dds.builtin_type.keyed_string.max_size	Maximum size of the strings published by the *DataWriters* and received by the *DataReaders* belonging to a *DomainParticipant* using the built-in type (includes the NULL-terminated character). Default: 1024
Octets	dds.builtin_type.octets.max_size	Maximum size of the octet sequences published by the *DataWriters* and *DataReaders* belonging to a *DomainParticipant*. Default: 2048
Keyed-Octets	dds.builtin_type.keyed_octets.max_key_size	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.
	dds.builtin_type.keyed_octets.max_size	Maximum size of the octet sequences published by the *DataWriters* and *DataReaders* belonging to a *DomainParticipant*. Default: 2048

application: a class or structure with fields, methods, and other members that you interact with directly.

In C and C++, you create and delete your own objects from factories, just as you create *RTI Data Distribution Service* 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.

```
/* 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 C++, as in C, 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);
```

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.

```
// 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.

```
// In the generated code:
public class MyData {
   public String myString = "";
}

// In your code:
MyData sample = new MyData();
String str = sample->myString; // empty, non-null string
```

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:

```
struct MyData {
   long 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 Defining New Types (Section 3.8.2).

For more information about the DynamicData and DynamicDataTypeSupport classes, consult the online API documentation.

In 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 C++:

```
DDSDynamicDataTypeSupport* 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 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_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.
 */
```

Chapter 4 DDS Entities

In DDS, the main classes extend an abstract base class called an *Entity* (see Figure 4.1). Every *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. *Entities* may also have attached *Conditions*, which provide a way to wait for status changes.

This chapter describes the common operations and general designed patterns shared by all *Entities* including *DomainParticipant*s, *Topics*, *Publisher*s, *DataWriter*s, *Subscriber*s, and *DataReader*s. In subsequent chapters, the specific statuses, *Listeners*, *Conditions*, and QosPolicies for each class will be discussed in detail.

This chapter includes the following sections:

- ❏ Common Operations for All DDS Entities (Section 4.1)
- ❏ QosPolicies (Section 4.2)
- ❏ Statuses (Section 4.3)
- ❏ Listeners (Section 4.4)
- ❏ Exclusive Areas (EAs) (Section 4.5)
- ❏ Conditions and WaitSets (Section 4.6)

4.1 Common Operations for All DDS Entities

All *Entities* (*DomainParticipant*s, *Topics*, *Publisher*s, *DataWriter*s, *Subscriber*s, and *DataReader*s) provide operations for:

- ❏ Creating and Deleting Entities (Section 4.1.1)
- ❏ Enabling Entities (Section 4.1.2)
- ❏ Getting an Entity's Instance Handle (Section 4.1.3)
- ❏ Getting Status and Status Changes (Section 4.1.4)
- ❏ Getting and Setting Listeners (Section 4.1.5)
- ❏ Getting the StatusCondition (Section 4.1.6)
- ❏ Getting and Setting QosPolicies (Section 4.1.7)

4.1.1 Creating and Deleting Entities

In DDS, the factory design pattern is used in creating and deleting *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.

Table 4.1 **DDS Entity Factories**

DDS Entity	Created by
DomainParticipant	DomainParticipantFactory (a static singleton object provided by *RTI Data Distribution Service*)
Topic	DomainParticipant
Publisher	
Subscriber	
DataWriter[a]	
DataReader[a]	
DataWriter[a]	Publisher
DataReader[a]	Subscriber

a. DataWriters may be created by a DomainParticipant or a Publisher. Similarly, DataReaders may be created by a DomainParticipant or a Subscriber.

All entities that are factories have:

- ❑ Operations to create and delete child entities. For example:

  DDSPublisher::create_datawriter,
  DDSDomainParticipant::delete_topic

- ❑ Operations to get and set the default QoS values used when creating child entities. For example:

  DDSSubscriber::get_default_datareader_qos,
  DDSDomainParticipantFactory::set_default_participant_qos

- ❑ An ENTITYFACTORY QosPolicy (Section 6.4.2) to specify whether or not the newly created child entity should be automatically enabled upon creation.

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()**.

4.1.2 Enabling 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 ENTITYFACTORY QosPolicy (Section 6.4.2) 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 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:

```
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*:

```
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, regardless of the setting in the factory's EntityFactoryQoS.
- ❑ If the factory is enabled, its children will be created either enabled or disabled, according to the setting in the factory's EntityFactory Qos.
- ❑ 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_RETCODE_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 *DomainParticipant*s until the entity is enabled (see Chapter 12: Discovery).

- ❑ 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 *RTI Data Distribution Service* 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).
- ❑ **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

4. DDS Entities

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.

Note: The builtin transports are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataWriter/DataReader* is created, or (c) you look up a builtin data reader, whichever happens first. Any changes to the builtin transport properties that are made after the builtin 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);
}
```

See Section 4.4 for more information about *statuses*.

4.1.5 Getting and Setting Listeners

Each type of *Entity* has an associated *Listener*, see Listeners (Section 4.4). 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 Section 4.4 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 Section 4.6 for more information about *StatusConditions* and *WaitSets*.

4.1.7 Getting and Setting QosPolicies

Each type of *Entity* has an associated set of QosPolicies (see Section 4.2). QosPolicies allow you to configure and set properties for the Entity.

While most QosPolicies are defined by the DDS specification, some are offered by *RTI Data Distribution Service* 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 Chapter 15: Configuring QoS with XML.

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 *DataReader*, 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.

Each QosPolicy has default values (listed in the online 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 Section 4.1.7.1.
- ❑ During Entity creation (if custom values are only needed for a particular Entity). See Section 4.1.7.2.
- ❑ After Entity creation (if the values initially specified for a particular Entity are no longer appropriate). See Section 4.1.7.3.

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. *RTI Data Distribution Service*-extended QosPolicies either cannot be changed after creation or can be changed at any time. The changeability of each QosPolicy is documented in the online documentation as well as in Table 4.2. 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 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.

```
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);
```

Note: 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 Chapter 15: Configuring QoS with XML.

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.

For example, to change the RELIABLE QosPolicy for a *DataWriter* before creating it:

```
// 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_BEST_EFFORT_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 Chapter 15: Configuring QoS with XML.

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*:

```
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 Chapter 15: Configuring QoS with XML.

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 Values

RTI Data Distribution Service 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 *RTI Data Distribution Service* default values:

- ❑ **DDS_PARTICIPANT_QOS_DEFAULT**
- ❑ **DDS_PUBLISHER_QOS_DEFAULT**
- ❑ **DDS_SUBSCRIBER_QOS_DEFAULT**
- ❑ **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 *RTI Data Distribution Service*-specified default values:

```
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 *RTI Data Distribution Service*-specified default values:

```
publisher->set_default_datawriter_qos(DDS_DATAWRITER_QOS_DEFAULT);
```

Note: These defaults *cannot* be used to initialize a QoS structure for an entity. For example, the following is NOT allowed:

Not Allowed—>

```
DataWriterQos dataWriterQos = DATAWRITER_QOS_DEFAULT;
// modify QoS...
create_datawriter(dataWriterQos);
```

4.2 QosPolicies

RTI Data Distribution Service'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 on page 4-13 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 Controlling Behavior with Quality of Service (QoS) Policies (Section 2.5.1), 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 would result in an inconsistent set of policies. To indicate a failure, **set_qos()** will return **INCONSISTENT_POLICY**. Section 4.2.1 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 on page 4-13.

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**

QosPolicy	Summary
Asynchronous-Publisher	Configures the mechanism that sends user data in an external middleware thread. See Section 6.4.1.
Batch	Specifies and configures the mechanism that allows *RTI Data Distribution Service* to collect multiple 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 Section 6.5.1.
Database	Various settings and resource limits used by *RTI Data Distribution Service* to control its internal database. See Section 8.5.1.
DataReaderProtocol	This QosPolicy configures the DDS on-the-network protocol, RTPS. See Section 7.6.1.
DataReader-ResourceLimits	Various settings that configure how *DataReaders* allocate and use physical memory for internal resources. See Section 7.6.2.
DataWriterProtocol	This QosPolicy configures the DDS on-the-network protocol, RTPS. See Section 6.5.2.
DataWriterResourceLimits	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 Section 6.5.3.
Deadline	For a *DataReader*, specifies the maximum expected elapsed time between arriving data samples. For a *DataWriter*, specifies a commitment to publish samples with no greater elapsed time between them. See Section 6.5.4.
DestinationOrder	Controls how *RTI Data Distribution Service* 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 Section 6.5.5.
Discovery	Configures the mechanism used by *RTI Data Distribution Service* to automatically discover and connect with new remote applications. See Section 8.5.2.

4. DDS Entities

Table 4.2 **QosPolicies**

QosPolicy	Summary
DiscoveryConfig	Controls the amount of delay in discovering entities in the system and the amount of discovery traffic in the network. See Section 8.5.3.
DomainParticipant-ResourceLimits	Various settings that configure how *DomainParticipants* allocate and use physical memory for internal resources, including the maximum sizes of various properties. See Section 8.5.4.
Durability	Specifies whether or not *RTI Data Distribution Service* will store and deliver data that were previously published to new *DataReaders*. See Section 6.5.6.
DurabilityService	Various settings to configure the external Persistence Service used by *RTI Data Distribution Service* for *DataWriters* with a Durability QoS setting of Persistent Durability. See Section 6.5.7.
EntityFactory	Controls whether or not child entities are created in the enabled state. See Section 6.4.2.
EntityName	Assigns a name to a *DomainParticipant*. See Section 8.5.5.
Event	Configures the *DomainParticipant's* internal thread that handles timed events. See Section 8.5.6.
ExclusiveArea	Configures multi-thread concurrency and deadlock prevention capabilities. See Section 6.4.3.
GroupData	Along with TOPIC_DATA QosPolicy (Section 5.2.1) and USER_DATA QosPolicy (Section 6.5.23), this QosPolicy is used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data. See 6.4.4.
History	Specifies how much data must to stored by *RTI Data Distribution Service*for the *DataWriter* or *DataReader*. This QosPolicy affects the RELIABILITY QosPolicy (Section 6.5.17) as well as the DURABILITY QosPolicy (Section 6.5.6). See Section 6.5.8.
LatencyBudget	Suggestion to DDS on how much time is allowed to deliver data. See Section 6.5.9.
Lifespan	Specifies how long *RTI Data Distribution Service* should consider data sent by an user application to be valid. See Section 6.5.10.
Liveliness	Specifies and configures the mechanism that allows *DataReaders* to detect when *DataWriter*s become disconnected or "dead." See Section 6.5.11.
MultiChannel	Configures a *DataWriter's* ability to send data on different multicast groups (addresses) based on the value of the data. See Section 6.5.12.
Ownership	Along with Ownership Strength, specifies if *DataReaders* for a topic can receive data from multiple *DataWriters* at the same time. See Section 6.5.13.
OwnershipStrength	Used to arbitrate among multiple *DataWriters* of the same instance of a Topic when Ownership QoSPolicy is EXLUSIVE. See Section 6.5.14.
Partition	Adds string identifiers that are used for matching *DataReaders* and *DataWriters* for the same *Topic*. See Section 6.4.5.
Presentation	Controls how DDS presents data received by an application to the *DataReaders* of the data. See Section 6.4.6.

Table 4.2 **QosPolicies**

QosPolicy	Summary
Profile	Configures the way that XML documents containing QoS profiles are loaded by RTI. See Section 8.4.1.
Property	Stores name/value(string) pairs that can be used to configure certain parameters of *RTI Data Distribution Service* 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 Section 6.5.15.
PublishMode	Specifies how *RTI Data Distribution Service* 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 *RTI Data Distribution Service* to use its own thread to send the data. See Section 6.5.16.
ReaderDataLife-Cycle	Controls how a *DataReader* manages the lifecycle of the data that it has received. See Section 7.6.3.
ReceiverPool	Configures threads used by *RTI Data Distribution Service* to receive and process data from transports (for example, UDP sockets). See Section 8.5.7.
Reliability	Specifies whether or not DDS will deliver data reliably. See Section 6.5.17.
ResourceLimits	Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See Section 6.5.18.
SystemResource-Limits	Configures *DomainParticipant*-independent resources used by *RTI Data Distribution Service*. Mainly used to change the maximum number of *DomainParticipants* that can be created within a single process (address space). See Section 8.4.2.
TimeBasedFilter	Set by a *DataReader* to limit the number of new data values received over a period of time. See Section 7.6.4.
TopicData	Along with Group Data QosPolicy and User Data QosPolicy, used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data. See Section 5.2.1.
TransportBuiltin	Specifies which built-in transport plugins are used. See Section 8.5.8.
TransportMulticast	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 Section 7.6.5.
TransportPriority	Set by a *DataWriter* to tell DDS that the data being sent is a different "priority" than other data. See Section 6.5.19.
TransportSelection	Allows you to select which physical transports a *DataWriter* or *DataReader* may use to send or receive its data. See Section 6.5.20.
TransportUnicast	Specifies a subset of transports and port number that can be used by an Entity to receive data. See Section 6.5.21.
TypeSupport	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 Section 6.5.22.

Table 4.2 **QosPolicies**

QosPolicy	Summary
UserData	Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data. See Section 6.5.23.
WireProtocol	Specifies IDs used by the RTPS wire protocol to create globally unique identifiers. See Section 8.5.9.
WriterDataLifeCycle	Controls how a *DataWriter* handles the lifecycle of the instances (keys) that the *DataWriter* is registered to manage. See Section 6.5.24.

4.2.1 QoS Requested vs. Offered Compatibility—the RxO Property

Some QosPolicies 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 *RTI Data Distribution Service* to connect the sending entity to the receiving entity, the offered behavior must satisfy the requested behavior.

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 Section 6.5.4).

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 OWNERSHIP QosPolicy (Section 6.5.13)).

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, *RTI Data Distribution Service* will compare the values of those policies for compatibility. If they are compatible, then *RTI Data Distribution Service* will connect the sending entity to the receiving entity allowing data to be sent between them. If they are found to be incompatible, then *RTI Data Distribution Service* will not interconnect the entities preventing data to be sent between them.

In addition, *RTI Data Distribution Service* will record this event by changing the associated communication status in both the sending and receiving applications, see Types of Communication Status (Section 4.3.1). Also, if you have installed *Listeners* on the associated *Entities*, then *RTI Data Distribution Service* will invoke the associated callback functions to notify user code that an incompatible QoS combination has been found, see Types of Listeners (Section 4.4.1).

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, *RTI Data Distribution Service* 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, *RTI Data Distribution Service* 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.

```
struct DDS_<Entity>Qos qos = DDS_<Entity>Qos_INITIALIZER;
```

- **DDS_<Entity>Qos_initialize()** This is a function that can be used to initialize a **DDS_<Entity>Qos** structure instead of the macro above.

```
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.

```
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.

```
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 *RTI Data Distribution Service* 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_<statusname>_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 *RTI Data Distribution Service* threads will call the listener methods when the status changes. See Listeners (Section 4.4).

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 Conditions and WaitSets (Section 4.6).

This section includes the following:

- ❑ Types of Communication Status (Section 4.3.1)
- ❑ Special Status-Handling Considerations for C (Section 4.3.2)

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 *RTI Data Distribution Service* is provided in Table 4.3 on page 4-20. 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).

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.
- ❑ **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 are *read* communications statuses: **DATA_AVAILABLE** and **DATA_ON_READERS**.

4. DDS Entities

Table 4.3 **Communication Statuses**

Related Entity	Status (DDS_*_STATUS)	Description	Reference
Topic	INCONSISTENT_TOPIC	Another *Topic* exists with the same name but different characteristics–for example a different type.	Section 5.3.1
DataWriter	DATA_WRITER_CACHE	The status of the *DataWriter's* cache. This status does not have a Listener.	Section 6.3.5.1
	DATA_WRITER_PROTOCOL	The status of a *DataWriter's* internal protocol related metrics (such as the number of samples pushed, pulled, filtered) and the status of wire protocol traffic. This status does not have a Listener.	Section 6.3.5.2
	LIVELINESS_LOST	The liveliness that the *DataWriter* has committed to (through its Liveliness QosPolicy) was not respected (assert_liveliness() or write() not called in time), thus *DataReader* entities may consider the *DataWriter* as no longer active.	Section 6.3.5.3
	OFFERED_DEADLINE_MISSED	The deadline that the *DataWriter* has committed through its Deadline QosPolicy was not respected for a specific instance of the *Topic*.	Section 6.3.5.4
	OFFERED_INCOMPATIBLE_QOS	An offered QosPolicy value was incompatible with what was requested by a *DataReader* of the same Topic.	Section 6.3.5.5
	PUBLICATION_MATCHED	The *DataWriter* found a *DataReader* that matches the *Topic*, has compatible QoSs and a common partition, or a previously matched *DataReader* has been deleted.	Section 6.3.5.6
	RELIABLE_WRITER_CACHE_CHANGED	The number of unacknowledged samples in a reliable *DataWriter's* cache has reached one of the predefined trigger points.	Section 6.3.5.7
	RELIABLE_READER_ACTIVITY_CHANGED	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*.	Section 6.3.5.8

Table 4.3 **Communication Statuses**

Related Entity	Status (DDS_*_STATUS)	Description	Reference
Subscriber	DATA_ON_READERS	New data is available for any of the readers that were created from the *Subscriber*.	Section 7.2.8
Data-Reader	DATA_AVAILABLE	New data (one or more samples) are available for the specific *DataReader*.	Section 7.3.6.1
	DATA_READER_CACHE	The status of the reader's cache. This status does not have a Listener.	Section 7.3.6.2
	DATA_READER_PROTOCOL	The status of a *DataReader's* internal protocol related metrics (such as the number of samples received, filtered, rejected) and the status of wire protocol traffic. This status does not have a Listener.	Section 7.3.6.3
	LIVELINESS_CHANGED	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*.	Section 7.3.6.4
	REQUESTED_DEADLINE_MISSED	New data was not received for an instance of the *Topic* within the time period set by the *DataReader's* Deadline QosPolicy.	Section 7.3.6.5
	REQUESTED_INCOMPATIBLE_QOS	A requested QosPolicy value was incompatible with what was offered by a *DataWriter* of the same *Topic*.	Section 7.3.6.6
	SAMPLE_LOST	A sample sent by *RTI Data Distribution Service* has been lost (never received).	Section 7.3.6.7
	SAMPLE_REJECTED	A received sample has been rejected due to a resource limit (buffers filled).	Section 7.3.6.8
	SUBSCRIPTION_MATCHED	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.	Section 7.3.6.9

As mentioned in Section 4.1.4, 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_READER**.

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*.

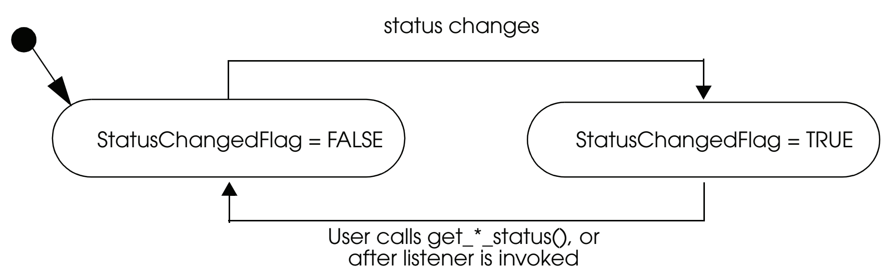
Both types of status have a *StatusChangedFlag*. This flag indicates whether that particular communication status has changed since the last time the status was read by the application. The way the *StatusChangedFlag* is maintained is slightly different for the *plain* communication status and the *read* communication status, as described in the following sections:

- ❑ Changes in Plain Communication Status (Section 4.3.1.1)
- ❑ Changes in Read Communication Status (Section 4.3.1.2)

4.3.1.1 Changes in Plain Communication Status

As seen in Figure 4.1 on page 4-22, 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_<plain communication status>()** operation.

Figure 4.1 **Status Changes for Plain Communication Status**



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_<plain communication 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 Types of Listeners (Section 4.4.1).)

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 4.2 on page 4-25, for the read communication status, the StatusChangedFlag flag is initially set to FALSE. The StatusChangedFlag becomes TRUE when either a data sample arrives or else the ViewStateKind, SampleStateKind, or InstanceStateKind of any existing 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:

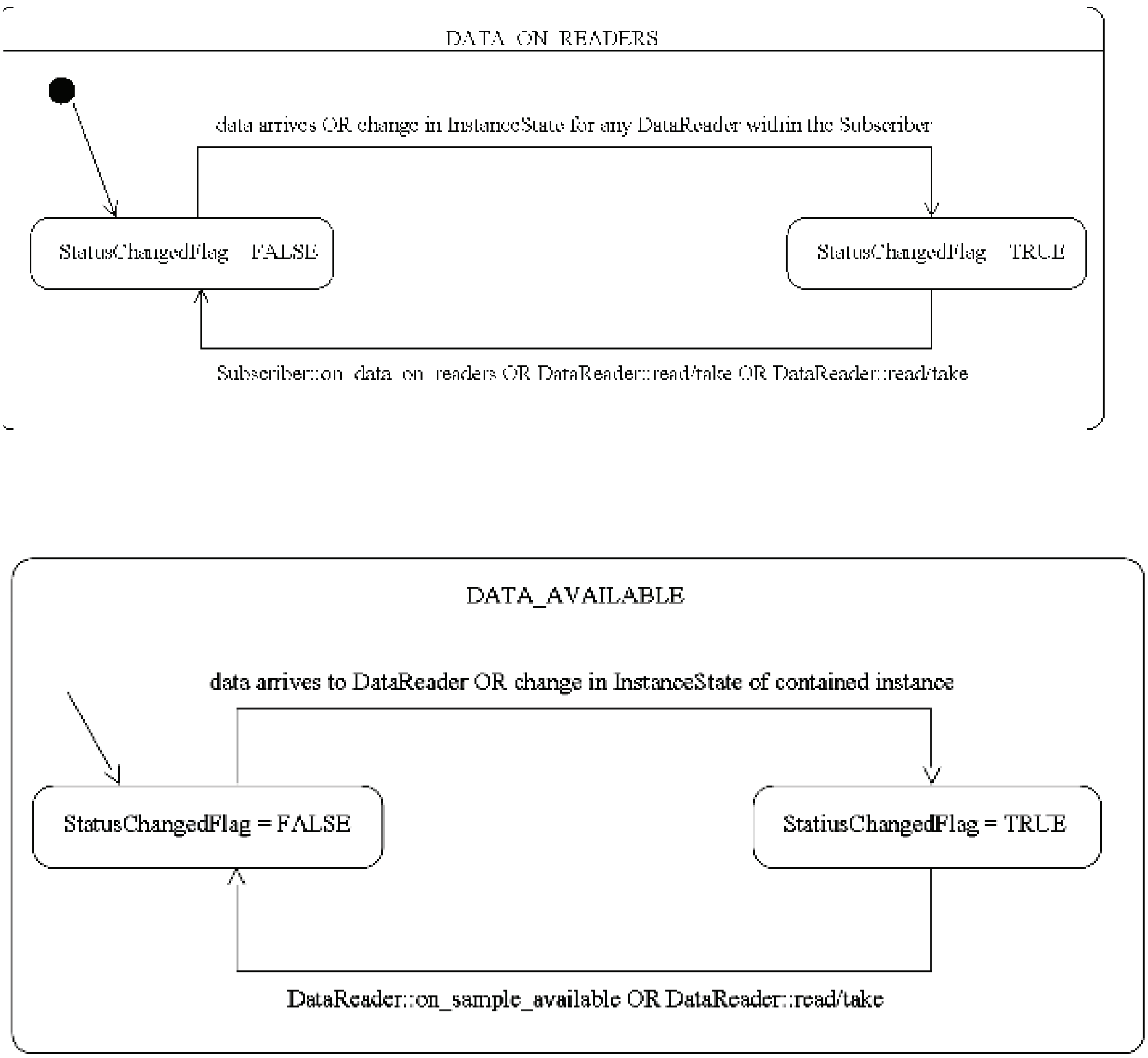
- ❏ The arrival of new data.
- ❏ A change in the InstanceStateKind of a contained instance. This can be caused by either:
  - Notification that an instance has been disposed by:
    - the *DataWriter* that owns it, if OWNERSHIP = EXCLUSIVE
    - or by any *DataWriter*, if OWNERSHIP = SHARED
  - The loss 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 again 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*.
    - One of the read/take operations (or their variants) is called on any *DataReader* belonging to the *Subscriber*.

Figure 4.2 **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, *RTI Data Distribution Service* 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, *RTI Data Distribution Service* will provide:

- ❑ **DDS_<STATUS>STATUS_INITIALIZER** This is a macro that should be used when a **DDS_<Status>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>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>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>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

This section describes *Listeners* and how to use them.

Listeners are triggered by changes in an entity's status. For instance, maybe *RTI Data Distribution Service* found a matching *DataReader* for a *DataWriter*, or new data has arrived for a *DataReader*.

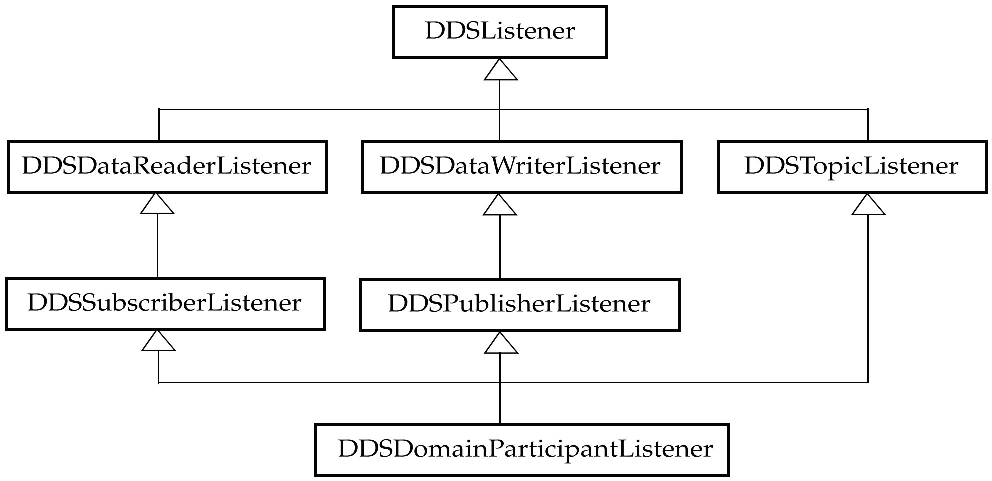
This section includes:

- Types of Listeners (Section 4.4.1)
- Creating and Deleting Listeners (Section 4.4.2)
- Special Considerations for Listeners in C (Section 4.4.3)
- Hierarchical Processing of Listeners (Section 4.4.4)
- Operations Allowed within Listener Callbacks (Section 4.4.5)

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 4.3. The methods are called by an internal *RTI Data Distribution Service* thread when the corresponding status for the *Entity* changes value.

Figure 4.3 **DDS 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:

```
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
                      DDS_DATA_AVAILABLE_STATUS;

datareader = subscriber->create_datareader(topic,
                                           DDS_DATAREADER_QOS_DEFAULT,
                                           listener, mask);
```

or afterwards:

```
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 describes what happens when a status change occurs. See Hierarchical Processing of Listeners (Section 4.4.4) for more information.

Table 4.4 **Effect of Different Combinations of Listeners and Status Bit Masks**

	No Bits Set in Mask	Some/All Bits Set in Mask
Listener is Specified	*RTI Data Distribution Service* finds the next most relevant listener for the changed status.	For the statuses that are enabled in the mask, the most relevant listener will be called. The 'statusChangedFlag' for the relevant status is reset.
Listener is NULL	*RTI Data Distribution Service* behaves as if the listener is not installed and finds the next most relevant listener for that status.	For the statuses that are enabled in the mask, *RTI Data Distribution Service* behaves as if the listener callback *is* installed, but the callback is doing nothing. This is called a 'nil' listener. The 'statusChangedFlag' for the relevant status is *not* reset.

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:

```
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

```
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 Hierarchical Processing of Listeners

As seen in Figure 4.3 on page 4-28, DDS *Listeners* for some entities derive from the 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. *RTI Data Distribution Service* 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 Processing Read Communication Statuses (Section 4.4.4.1).

For example, suppose that *RTI Data Distribution Service* 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 lists the callback functions that are available for each *Entity*'s status listener.

Table 4.5 **DDS Listener Callback Functions**

Entity Listener for:		Callback Functions
DomainParticipants	Topics	on_inconsistent_topic()
	Publishers and DataWriters	on_liveliness_lost()
		on_offered_deadline_missed()
		on_offered_incompatible_qos()
		on_publication_matched()
		on_reliable_reader_activity_changed()
		on_reliable_writer_cache_changed()
	Subscribers	on_data_on_readers()
	Subscribers and DataReaders	on_data_available
		on_liveliness_changed()
		on_requested_deadline_missed()
		on_requested_incompatible_qos()
		on_sample_lost()
		on_sample_rejected()
		on_subscription_matched()

4.4.4.1 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();
}
```

4.4.5 Operations Allowed within Listener Callbacks

Due to the potential for deadlock, some *RTI Data Distribution Service* APIs should not be invoked within the functions of listener callbacks. Exactly which *RTI Data Distribution Service* APIs are restricted depends on the *Entity* upon which the *Listener* is installed, as well as the configuration of 'Exclusive Areas,' as discussed in Section 4.5.

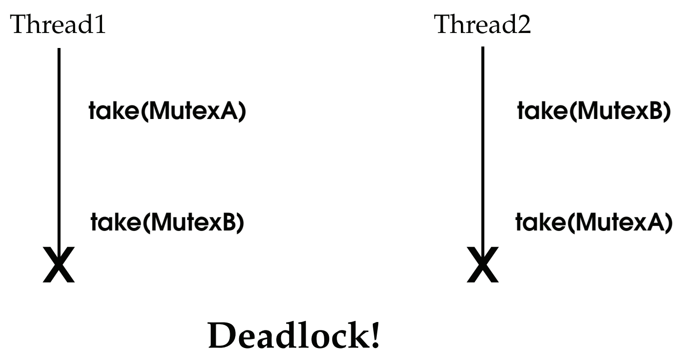
Please read and understand Exclusive Areas (EAs) (Section 4.5) and Restricted Operations in Listener Callbacks (Section 4.5.1) to ensure that the calls made from your *Listeners* are allowed and will not cause potential deadlock situations.

4.5 Exclusive Areas (EAs)

Listener callbacks are invoked by internal *RTI Data Distribution Service* 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 *RTI Data Distribution Service* API. Internally, those *RTI Data Distribution Service* 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 4.4's scenario, in which there are two threads and two mutexes.

Figure 4.4 **Multiple Mutexes Leading to a Deadlock Condition**



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 4.4 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. *RTI Data Distribution Service* has been carefully created and analyzed so that we know our threads internally are safe from deadlock interactions.

However, when *RTI Data Distribution Service* threads that are holding mutexes call user code in listeners, it is possible for user code to inadvertently cause the threads to deadlock if *RTI Data Distribution Service* 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 *RTI Data Distribution Service* APIs within user callback code, and a QoS policy that allows you to configure *Exclusive Areas*.

RTI Data Distribution Service 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 *RTI Data Distribution Service*

4. DDS Entities

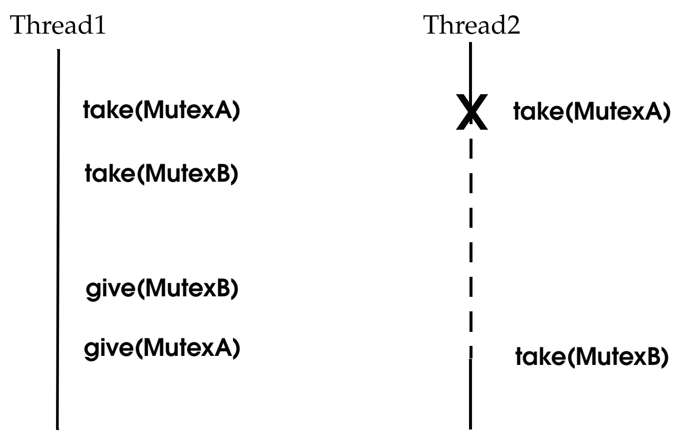
has an associated EA, EAs may be shared among several entities. A thread is automatically in the entity's EA when it is calling the entity's listener.

RTI Data Distribution Service allows you to configure all the *Entities* within an application in a single domain to share a single *Exclusive Area*. This would greatly restrict the concurrency of thread execution within *RTI Data Distribution Service*'s multi-threaded core. However, doing so would release all restrictions on using *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 4.4. 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 4.5.

Figure 4.5 **Taking Multiple Mutexes in a Specific Order to Eliminate Deadlock**



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.

RTI Data Distribution Service 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
- **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, <u>operations on *DataReaders* of the same *Subscriber*, will be serialized</u>, 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, <u>operations on *DataWriters* of the same *Publisher* will be serialized</u>, 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 *RTI Data Distribution Service* 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 Exclusive Areas (EAs) (Section 4.5), this section describes how EAs restrict you from using various *RTI Data Distribution Service* APIs from within the Listener callbacks of different *Entities*.

Note: 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:

- create any entities
- delete any entities
- enable any entities
- set QoS's 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*

RTI Data Distribution Service 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 EXCLUSIVE_AREA QosPolicy (DDS Extension) (Section 6.4.3) 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 children 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*, *Subscriber*s or *DomainParticipants*.

4.6 Conditions and WaitSets

Conditions and *WaitSets* provide another way for *RTI Data Distribution Service* 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 else 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:

- GuardConditions (Section 4.6.6) 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()**. *RTI Data Distribution Service* does not trigger or clear this type of condition—it is completely controlled by your application.
- ReadConditions and QueryConditions (Section 4.6.7) are created by your application, but triggered by *RTI Data Distribution Service*. *ReadCondition*s provide a way for you to specify the data samples that you want to wait for, by indicating the desired sample-states, view-states, and instance-states[1].
- StatusConditions (Section 4.6.8) are created automatically by *RTI Data Distribution Service*, one for each *Entity*. A *StatusCondition* is triggered by *RTI Data Distribution Service* when there is a change to any of that *Entity's* enabled statuses.

Figure 4.6 on page 4-39 shows the relationship between these objects and other *Entities* in the system.

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).

For example, to delete a WaitSet:

```
delete waitset;
```

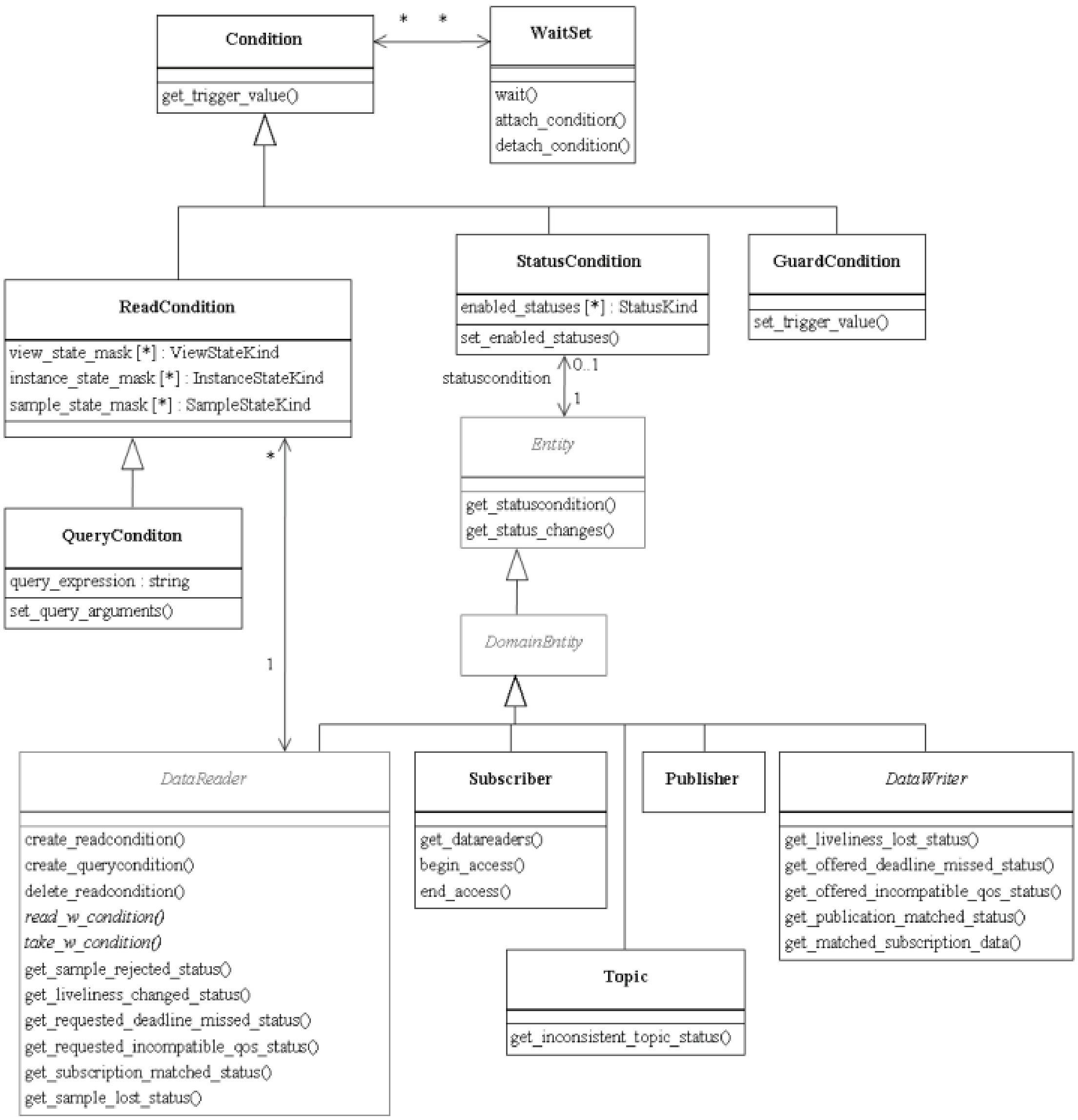
There are two ways to create a *WaitSet*—with or without specifying *WaitSet* properties (**DDS_WaitSetProperty_t**, described in Table 4.6).

- 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* is 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 sched-

1. These states are described in The SampleInfo Structure (Section 7.4.5).

Figure 4.6 **Conditions and WaitSets**



4. DDS Entities

Table 4.6 **WaitSet Properties (DDS_WaitSet_Property_t)**

Type	Field Name	Description
long	max_event_count	Maximum number of trigger events to cause a *WaitSet* to wake up.
DDS_Duration_t	max_event_delay	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.

ule 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.

To create a WaitSet with default behavior:

```
WaitSet* waitset = new WaitSet();
```

❑ If properties *are* specified when the *WaitSet* is created, the *WaitSet* will wait for either (a) up to **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()**.

To create a WaitSet with properties:

```
DDS_WaitSetProperty_t prop;
Prop.max_event_count = 5;
DDSWaitSet* waitset = new DDSWaitSet(prop);
```

4.6.2 WaitSet Operations

WaitSets have only a few operations, as listed in Table 4.7 on page 4-41. For details, see the online documentation.

Table 4.7 **WaitSet Operations**

Operation	Description
attach_condition	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, *RTI Data Distribution Service* will return an OUT_OF_RESOURCES error code.
detach_condition	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.
wait	Blocks execution of the thread until one or more attached *Conditions* becomes true, or until a user-specified timeout expires. See Section 4.6.3.
get_conditions	Retrieves a list of attached *Conditions*.
get_property	Retrieves the DDS_WaitSetProperty_t structure of the associated WaitSet.
set_property	Sets the DDS_WaitSetProperty_t structure, to configure the associated WaitSet to return after one or more trigger events have occurred.

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 TRUE, **wait()** blocks—suspending the calling thread. The **wait()** call will return when either (a) one or more of the attached *Conditions* becomes TRUE or (b) a user-specified timeout period expires.

You can also configure the properties of the *WaitSet* so that it will wait for up to **max_event_count** trigger events to occur before returning, or for up to **max_event_delay** time from the occurrence of the first trigger event before returning. See Creating and Deleting WaitSets (Section 4.6.1).

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.

Note: 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 4.7. 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, *RTI Data Distribution Service* will only unblock the *WaitSet* once.

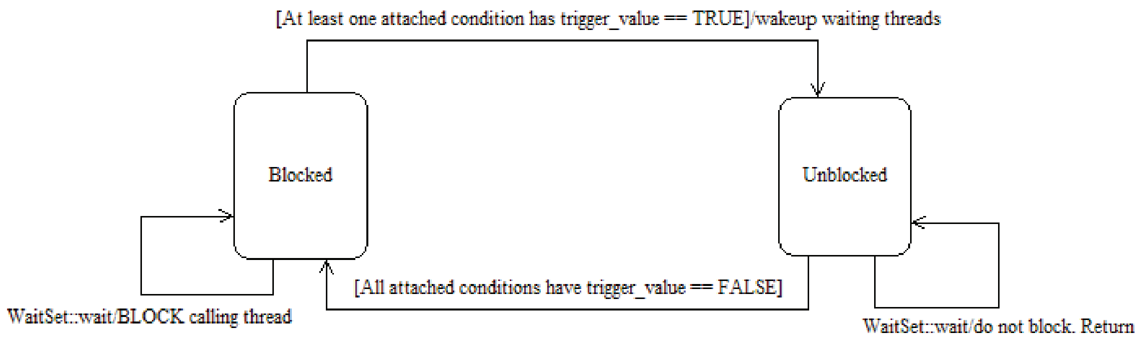
4.6.4 Processing Triggered Conditions—What to do when Wait() Returns

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.

Figure 4.7 **WaitSet Blocking Behavior**



- 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 DATA_AVAILABLE: 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 read_w_condition and take_w_condition (Section 7.4.3.6)).

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 Conditions and WaitSet Example (Section 4.6.5) to see how to determine which of the attached *Conditions* is in the returned list.

1. And then read/take on the returned *DataReader* objects.

4.6.5 Conditions and WaitSet Example

This example creates a *WaitSet* and then waits for one or more attached *Conditions* to become true.

```
// 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 = ...;
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;
           }
        }
   }
}

if (is_cond1_triggered) {
     // ... do something because "cond1" was triggered ...
}

if (is_cond2_triggered) {
     // ... do something because "cond2" was triggered ...
}
```

```
// Delete the waitset
delete waitset;
waitset = NULL;
```

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()**.

RTI Data Distribution Service 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:

```
// 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)**, *RTI Data Distribution Service* 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 *RTI Data Distribution Service*. *ReadCondition*s provide a way for you to specify the data samples that you want to wait for, by indicating the desired sample-states, view-states, and instance-states[1]. Then *RTI Data Distribution Service* will trigger the *ReadCondition* when suitable 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

1. These states are described in The SampleInfo Structure (Section 7.4.5).

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 to 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:

```
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);
```

Note: If you are using a *ReadCondition* to simply detect the presence of new data, consider using a *StatusCondition* (Section 4.6.8) with the DATA_AVAILABLE_STATUS instead, which will perform better in this situation.

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:

```
DDS_ReturnCode_t  delete_readcondition (DDSReadCondition *condition)
```

After a *ReadCondition* is triggered, use the *FooDataReader's* read/take "with condition" operations (see Section 7.4.3.6) to access the samples.

Table 4.8 lists the operations available on *ReadConditions*.

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 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 sample must be such that the **query_expression** evaluates to TRUE.

Table 4.8 **ReadCondition and QueryCondition Operations**

Operation	Description
get_datareader	Returns the *DataReader* to which the *ReadCondition* or *QueryCondition* is attached.
get_instance_state_mask	Returns the instance states that were specified when the *ReadCondition* or *QueryCondition* was created. These are the sample's instance states that *RTI Data Distribution Service* checks to determine whether or not to trigger the *ReadCondition* or *QueryCondition* .
get_sample_state_mask	Returns the sample-states that were specified when the *ReadCondition* or *QueryCondition* was created. These are the sample states that *RTI Data Distribution Service* checks to determine whether or not to trigger the *ReadCondition* or *QueryCondition.*
get_view_state_mask	Returns the view-states that were specified when the *ReadCondition* or *QueryCondition* was created. These are the view states that *RTI Data Distribution Service* checks to determine whether or not to trigger the *ReadCondition* or *QueryCondition.*

The **trigger_value** of a *ReadCondition* depends on the presence of 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 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 sample arrives and will transition to FALSE as soon as all the newly arrived samples are either read (so their status changes to READ) or taken (so they are no longer managed by *RTI Data Distribution Service*). 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 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 sample must also satisfy the conditions of the content filter associated with the *QueryCondition*.

Table 4.9 **QueryCondition Operations**

Operation	Description
get_query_expression	Returns the query expression specified when the *QueryCondition* was created.
get_query_parameters	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.
set_query_parameters	Changes the query parameters associated with the *QueryCondition*.

4.6.8 StatusConditions

StatusConditions are created automatically by *RTI Data Distribution Service*, one for each *Entity*. *RTI Data Distribution Service* will trigger the *StatusCondition* when there is a change to any of that *Entity's* enabled statuses.

By default, when *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 ***WaitSet***s is detailed in How StatusConditions are Triggered (Section 4.6.8.1).

Table 4.10 lists the operations available on *StatusConditions*.

Table 4.10 **StatusCondition Operations**

Operation	Description
set_enabled_statuses	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.
get_enabled_statuses	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
get_entity	Returns the *Entity* associated with the *StatusCondition*. Note that there is exactly one *Entity* associated with each *StatusCondition*.

Unlike other types of *Conditions*, *StatusConditions* are created by *RTI Data Distribution Service*, 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();
```

After a *StatusCondition* is triggered, call the *Entity's* **get_status_changes()** operation to see which status(es) changed.

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**==FALSE only if all the values of the **ChangedStatusFlags** are FALSE.

The sensitivity of the *StatusCondition* to a particular communication status is controlled by the list of **enabled_statuses** set on the *Condition* by means of the **set_enabled_statuses()** operation.

4.6.9 Using Both Listeners *and* WaitSets

You can use *Listeners* and *WaitSets* in the same application. For example, you may want to use *WaitSets* and *Conditions* to access the data, and *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, it both are enabled, then the *Listener* mechanism is used first, then the *WaitSet* objects are signalled.

Chapter 5 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 *RTI Data Distribution Service*. 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 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:

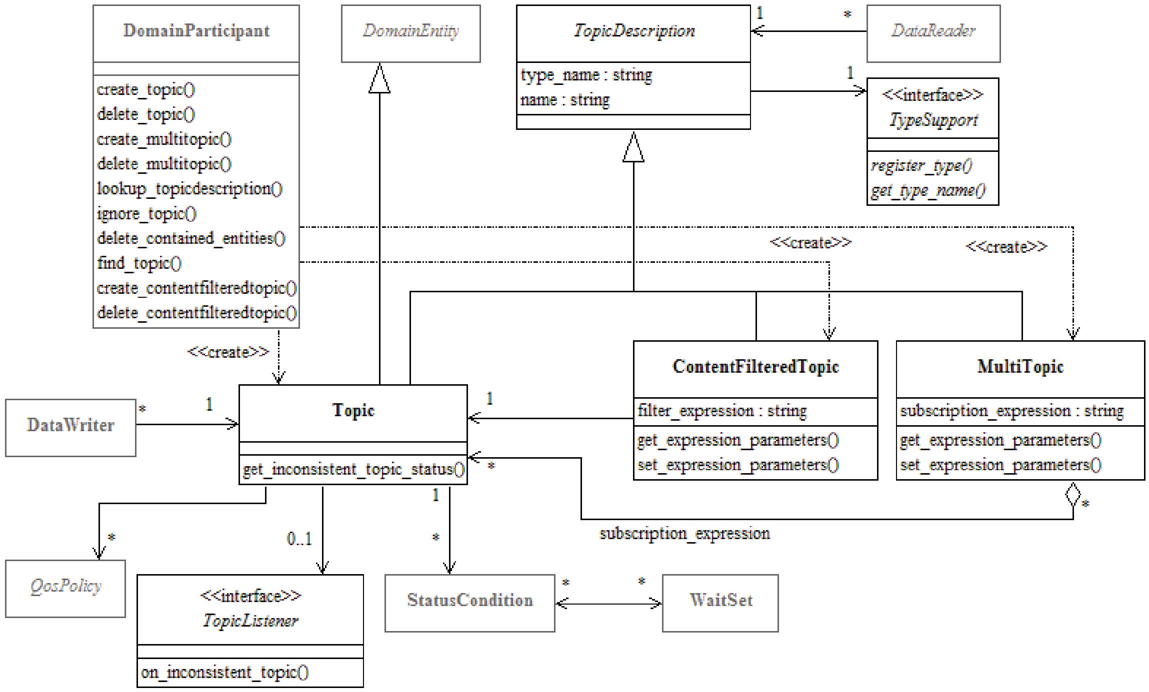
- Multiple *Topics* (each with a unique name) can use the same user data type.
- Applications may have multiple *DataWriters* for each *Topic*.
- Applications may have multiple *DataReaders* for each *Topic*.
- *DataWriters* and *DataReaders* must be associated with the same *Topic* in order for them 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).

This chapter includes the following sections:

- Topics (Section 5.1)
- Topic QosPolicies (Section 5.2)
- Status Indicator for Topics (Section 5.3)
- ContentFilteredTopics (Section 5.4)

Builtin Topics: *RTI Data Distribution Service* uses 'Builtin Topics' to discover and keep track of remote entities, such as new participants in the domain. Builtin Topics are discussed in Chapter 14.

Figure 5.1 **Topic Module**



Note: MultiTopics are not supported.

5.1 Topics

Before you can create a *Topic*, you need a user data type (see Chapter 3) and a *DomainParticipant* (Section 8.3). The user data type must be registered with the *DomainParticipant* (as we saw in the User Data Types chapter in Section 3.8.5.1).

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. For details on using these operations, see the reference section or the online documentation.

Table 5.1 **Topic Operations**

Purpose	Operation	Description	Reference
Configuring the Topic	enable	Enables the *Topic*.	Section 4.1.2
	get_qos	Gets the *Topic's* current QosPolicy settings. This is most often used in preparation for calling set_qos().	Section 5.1.3
	set_qos	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.	
	set_qos_with_profile	Sets the *Topic's* QoS based on a specified QoS profile.	
	get_listener	Gets the currently installed Listener.	Section 5.1.5
	set_listener	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*.	
	narrow	A type-safe way to cast a pointer. This takes a DDSTopicDescription pointer and 'narrows' it to a DDSTopic pointer.	Section 6.3.6
Checking Status	get_inconsistent_topic_status	Allows an application to retrieve a *Topic's* INCONSISTENT_TOPIC_STATUS status.	Section 5.3.1
	get_status_changes	Gets a list of statuses that have changed since the last time the application read the status or the listeners were called.	Section 4.1.4
Navigating Relationships	get_name	Gets the *topic_name* string used to create the *Topic*.	Section 5.1.1
	get_type_name	Gets the *type_name* used to create the *Topic*.	
	get_participant	Gets the *DomainParticipant* to which this *Topic* belongs.	Section 5.1.6.1

5.1.1 Creating Topics

Topics are created using the *DomainParticipant*'s **create_topic()** or **create_topic_with_profile()** operation:

```
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)
```

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 Chapter 15: Configuring QoS with XML.

topic_name Name for the new *Topic*, must not exceed 255 characters.

type_name Name for the user data type, must not exceed 255 characters. It must be the same name that was used to register the type, and the type must be registered with the same *DomainParticipant* used to create this *Topic*. See Section 3.6.

qos If you want to use the default QoS settings (described in the online documentation), use **DDS_TOPIC_QOS_DEFAULT** for this parameter (see Figure 5.2). If you want to customize any of the QosPolicies, supply a QoS structure (see Section 5.1.3).

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.

listener *Listeners* are callback routines. *RTI Data Distribution Service* 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 Section 5.1.5.

mask 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 Listeners (Section 4.4).

library_name A QoS Library is a named set of QoS profiles. See QoS Libraries (Section 15.9). If NULL is used for **library_name**, the *DomainParticipant's* default library is assumed.

profile_name A QoS profile groups a set of related QoS, usually one per entity. See QoS Profiles (Section 15.8). If NULL is used for **profile_name**, the *DomainParticipant's* default profile is assumed and **library_name** is ignored.

Note: It is not safe to create a topic while another thread is calling **lookup_topicdescription()** for that same topic (see Section 8.3.7).

Figure 5.2 **Creating a Topic with Default QosPolicies**

```
const char *type_name = NULL;

// register the type
type_name = FooTypeSupport::get_type_name();
retcode = FooTypeSupport::register_type(participant, type_name);
if (retcode != DDS_RETCODE_OK) {
    // handle error
}

// create the topic
DDSTopic* topic = participant->create_topic("Example Foo",
                                type_name, DDS_TOPIC_QOS_DEFAULT,
                                NULL, DDS_STATUS_MASK_NONE);
if (topic == NULL) {
    // process error here
};
```

For more examples, see Configuring QoS Settings when the Topic is Created (Section 5.1.3.1).

5.1.2 Deleting Topics

To delete a *Topic*, use the *DomainParticipant's* **delete_topic()** operation:

```
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.

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:

```
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;
} DDS_TopicQos;
```

Table 5.2 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 online documentation for each policy.

Table 5.2 **Topic QosPolicies**

QosPolicy	Description
Deadline	For a *DataReader*, specifies the maximum expected elapsed time between arriving data samples. For a *DataWriter*, specifies a commitment to publish samples with no greater elapsed time between them. See Section 6.5.4.
DestinationOrder	Controls how *RTI Data Distribution Service* 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 Section 6.5.5.

Table 5.2 **Topic QosPolicies**

QosPolicy	Description
Durability	Specifies whether or not *RTI Data Distribution Service* will store and deliver data that were previously published to new *DataReaders*. See Section 6.5.6.
DurabilityService	Various settings to configure the external Persistence Service used by *RTI Data Distribution Service* for *DataWriters* with a Durability QoS setting of Persistent Durability. See Section 6.5.7.
History	Specifies how much data must to stored by *RTI Data Distribution Service*for the *DataWriter* or *DataReader*. This QosPolicy affects the RELIABILITY QosPolicy (Section 6.5.17) as well as the DURABILITY QosPolicy (Section 6.5.6). See Section 6.5.8.
LatencyBudget	Suggestion to DDS on how much time is allowed to deliver data. See Section 6.5.9.
Lifespan	Specifies how long *RTI Data Distribution Service* should consider data sent by an user application to be valid. See Section 6.5.10.
Liveliness	Specifies and configures the mechanism that allows *DataReaders* to detect when *DataWriter*s become disconnected or "dead." See Section 6.5.11.
Ownership	Along with Ownership Strength, specifies if *DataReaders* for a topic can receive data from multiple *DataWriters* at the same time. See Section 6.5.13.
Reliability	Specifies whether or not DDS will deliver data reliably. See Section 6.5.17.
ResourceLimits	Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See Section 6.5.18.
TopicData	Along with Group Data QosPolicy and User Data QosPolicy, used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data. See Section 5.2.1.
TransportPriority	Set by a *DataWriter* to tell DDS that the data being sent is a different "priority" than other data. See Section 6.5.19.

5.1.3.1 Configuring QoS Settings when the Topic is Created

As described in Creating Topics (Section 5.1.1), 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 on page 5-5, 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 Section 8.3.6.4).

- ❏ To create a *Topic* with non-default QoS values, without using a QoS profile, use the *DomainParticipant*'s **get_default_topic_qos()** method 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 Changing QoS Settings After the Topic Has Been Created

There are 2 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 on page 5-9. 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.
- You can also change a *Topic's* (and all other Entities') QoS by using a QoS Profile. For an example, see Figure 5.4 on page 5-9. For more information, see Chapter 15: Configuring QoS with XML.

Figure 5.3 **Changing the QoS of an Existing Topic (without a QoS Profile)**

```
DDS_TopicQos topic_qos;[1]
// 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
}
```

1. For the C API, you need to use DDS_TopicQos_INITIALIZER or DDS_TopicQos_initialize(). See Special QosPolicy Handling Considerations for C (Section 4.2.2)

Figure 5.4 **Changing the QoS of an Existing Topic with a QoS Profile**

```
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 chapter, while the others are described in the sections noted in the Reference column of Table 5.2. The rest of the QosPolicies for a *Topic* can also be set on the corresponding *DataWriter*s and/or *DataReader*s. Actually, the values that *RTI Data Distribution Service* 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 Section 4.2.1).

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:

- ❏ 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 online 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), *RTI Data Distribution Service* 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 INCONSISTENT_TOPIC Status (Section 5.3.1).

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

If a *Topic*'s Listener has not been set and *RTI Data Distribution Service* detects an inconsistent *Topic*, the *DomainParticipantListener* (if it exists) will be notified instead (see Section 8.3.5). 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:

```
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 Type Name

If you want to retrieve the *topic_name* or *type_name* used in the **create_topic()** operation, use these methods:

```
const char* DDSTopicDescription::get_type_name();
const char* DDSTopicDescription::get_name();
```

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 on page 5-6.

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 Section 5.1.3); 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 Chapter 12: Discovery) using builtin-topics (see Chapter 14: Built-In Topics). How this information is used will be up to user code. *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 Chapter 14: Built-In Topics).

The structure for the TOPIC_DATA QosPolicy includes just one field, as seen in Table 5.3. 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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4).

This policy is similar to the GROUP_DATA (Section 6.4.4) and USER_DATA (Section 6.5.23) policies that apply to other types of Entities.

Table 5.3 **DDS_TopicDataQosPolicy**

Type	Field Name	Description
DDS_OctetSeq	value	default: empty

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 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 *RTI Data Distribution Service* to send packets containing the new TOPIC_DATA to all of the other applications in the 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 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

- ❏ GROUP_DATA QosPolicy (Section 6.4.4)
- ❏ USER_DATA QosPolicy (Section 6.5.23)
- ❏ DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)

5.2.1.4 Applicable Entities

- ❏ Topics (Section 5.1)

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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4). Because *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 (Section 6.4.4) and USER_DATA (Section 6.5.23) 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 Section 4.4 for a general discussion on Listeners and Statuses.

5.3.1 INCONSISTENT_TOPIC Status

In order for two applications to communicate with the same *Topic*, it has to be created with the same name and data type. This status indicates that another *DomainParticipant* has created a *Topic* using the same name as the local *Topic*, but with a different data type.

The status is a structure of type **DDS_InconsistentTopicStatus**, see Table 5.4. The **total_count** keeps track of the total number of times that an inconsistent topic was found. 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 *RTI Data Distribution Service*.

Table 5.4 **DDS_InconsistentTopicStatus Structure**

Type	Field Name	Description
DDS_Long	total_count	Total number of *Topics* discovered whose name matches the Topic to which this status is attached but whose type is inconsistent.
DDS_Long	total_count_change	The change in total_count since the last time this status was read.

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 data samples a *DataReader* has to process and may also reduce the amount of data sent over the network.

This section includes the following:

- ❏ Overview (Section 5.4.1)
- ❏ Where Filtering is Applied—Publishing vs. Subscribing Side (Section 5.4.2)
- ❏ Creating ContentFilteredTopics (Section 5.4.3)
- ❏ Deleting ContentFilteredTopics (Section 5.4.4)
- ❏ Using a ContentFilteredTopic (Section 5.4.5)
- ❏ SQL Filter Expression Notation (Section 5.4.6)
- ❏ Example SQL Filter Expressions (Section 5.4.6.11)
- ❏ Custom Content Filters (Section 5.4.8)

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* a DDS entity—it does not have QosPolicies or *Listeners*.

A ContentFilteredTopic relates to other entities in 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*.
- ❑ 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.)

RTI Data Distribution Service also supports network-switch filtering for multi-channel *DataWriters* (see Chapter 16: Multi-channel DataWriters).

A *DataWriter* will automatically filter 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 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3)). This value can be from 0-32. 0 means do not filter any *DataReader* and 32 (default value) means filter up to 32 *DataReaders*.
   - 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 4 matching *DataReaders* in the same locator (see Peer Descriptor Format (Section 12.2.1)).
4. The *DataWriter* has infinite liveliness. (See LIVELINESS QosPolicy (Section 6.5.11).)
5. The *DataWriter* is *not* using an Asynchronous Publisher. (That is, the *DataWriter's* PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.16) *kind* is set to DDS_SYNCHRONOUS_PUBLISHER_MODE_QOS.)
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 type-code information is sent on the wire (see Using Generated Types without RTI Data Distribution Service (Standalone) (Section 3.7)).

Note: In addition to filtering new samples, a *DataWriter* can also be configured to filter previously written samples stored in the *DataWriter*'s queue for newly discovered *DataReaders*. To do so, use the **refilter** field in the *DataWriter's* HISTORY QosPolicy (Section 6.5.8).

5.4.3 Creating ContentFilteredTopics

To create a ContentFilteredTopic that uses the default SQL filter, use the *DomainParticipant's* **create_contentfilteredtopic()** operation:

```
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 Section 5.4.7), use the **create_contentfilteredtopic_with_filter()** variation:

```
DDS_ContentFilteredTopic *create_contentfilteredtopic_with_filter
                        (const char           *  name,
                         DDSTopic             *  related_topic,
                         const char           *  filter_expression,
                         const DDS_StringSeq  &  expression_parameters,
                         const char           *  filter_name =
                                                    DDS_SQLFILTER_NAME)
```

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 sample is received; otherwise it is discarded. This parameter cannot be NULL. Once a ContentFilteredTopic is created, its **filter_expression** cannot be changed. The notation for this expression depends on the filter that you are using (specified by the **filter_name** parameter). See SQL Filter Expression Notation (Section 5.4.6) and STRINGMATCH Filter Expression Notation (Section 5.4.7).

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()** (Section 5.4.5.2), **append_to_expression_parameter()** (Section 5.4.5.3) and **remove_from_expression_parameter()** (Section 5.4.5.4).

filter_name Name of the content filter to use for filtering. The filter must have been previously registered with the *DomainParticipant* (see Registering a Custom Filter (Section 5.4.8.1)). There are two builtin filters, DDS_SQLFILTER_NAME[1] (the default filter) and DDS_STRINGMATCHFILTER_NAME[1]—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:
<field name> MATCH <string pattern> (see Section 5.4.7).

.If you run *rtiddsgen* with **-notypecode**, then you must use the "**with_filter**" version with a custom filter instead—do not use the builtin SQL filter or the STRINGMATCH filter with the **-notypecode** option because they require type-codes. See rtiddsgen Command-Line Arguments (Section 3.6.1).

To summarize:

❏ To use the builtin default SQL filter:
  - Do not use **-notypecode** when running *rtiddsgen*
  - Call **create_contentfilteredtopic()**
  - See SQL Filter Expression Notation (Section 5.4.6)

❏ To use the builtin STRINGMATCH filter:
  - Do not use **-notypecode** when running *rtiddsgen*
  - Call **create_contentfilteredtopic_with_filter()**, setting the **filter_name** to **DDS_STRINGMATCHFILTER_NAME**
  - See STRINGMATCH Filter Expression Notation (Section 5.4.7)

❏ To use a custom filter:
  - call **create_contentfilteredtopic_with_filter()**, setting the **filter_name** to a registered custom filter

❏ To use *rtiddsgen* with **-notypecode**:
  - call **create_contentfilteredtopic_with_filter()**, setting the **filter_name** to a registered custom filter

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.

Note: Be careful with memory management of the string sequence in some of the ContentFilteredTopic APIs. See the **String Support** section in the online documentation (within the **Infrastructure** module) for details on sequences.

5.4.4 Deleting ContentFilteredTopics

To delete a ContentFilteredTopic, use the *DomainParticipant's* **delete_contentfilteredtopic()** operation:

1. Make sure no *DataReaders* are using the ContentFilteredTopic. (If this is not true, the operation returns **PRECONDITION_NOT_MET**.)
2. Delete the ContentFilteredTopic by using the *DomainParticipant's* **delete_contentfilteredtopic()** operation.

```
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.

Table 5.5 **ContentFilteredTopic Operations**

Operation	Description	Reference
append_to_expression_parameter	Concatenates a string value to the input expression parameter	Section 5.4.5.3
get_expression_parameters	Gets the expression parameters.	Section 5.4.5.1
get_filter_expression	Gets the expression.	Section 5.4.5.5
get_related_topic	Gets the related Topic.	Section 5.4.5.6
narrow	Casts a DDS_TopicDescription pointer to a ContentFilteredTopic pointer.	Section 5.4.5.7
remove_from_expression_parameter	Removes a string value from the input expression parameter	Section 5.4.5.4
set_expression_parameters	Changes the expression parameters.	Section 5.4.5.2

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)
```

parameters The filter expression parameters.

The memory for the strings in this sequence is managed as described in the **String Support** section of the online documentation (within the **Infrastructure** module). In particular, be careful to avoid a situation in which *RTI Data Distribution Service* allocates a string on your behalf and you then reuse that string in such a way that *RTI Data Distribution Service* 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, if that was never called, the parameters specified when the ContentFilteredTopic was created.

5.4.5.2 Setting Expression Parameters

To change the expression parameters associated with a ContentFilteredTopic:

```
DDS_ReturnCode_t set_expression_parameters
                         (const struct DDS_StringSeq & parameters)
```

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.

Note: 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 online 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:

```
DDS_ReturnCode_t append_to_expression_parameter(
                         const DDS_Long index,
                         const char* value);
```

5. Topics

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 SQL Extension: Regular Expression Matching (Section 5.4.6.4)) to add a pattern to the match pattern list. For example, if **filter_expression** is:

```
symbol MATCH 'IBM'
```

Then **append_to_expression_parameter(0, "MSFT")** would generate the expression:

```
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 SQL Extension: Regular Expression Matching (Section 5.4.6.4)) to remove a pattern from the match pattern list. For example, if **filter_expression** is:

```
symbol MATCH 'IBM,MSFT'
```

Then **remove_from_expression_parameter(0, "IBM")** would generate the expression:

```
symbol MATCH 'MSFT'
```

5.4.5.5 Getting the Filter Expression

To get the filter expression that was specified when the ContentFilteredTopic was created:

```
const char* get_filter_expression ()
```

There is no corresponding **set** operation. The filter expression can only be set when the ContentFilteredTopic is created.

5.4.5.6 Getting the Related Topic

To get the related topic that was specified when the ContentFilteredTopic was created:

```
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:

```
DDS_TopicDescription* narrow ()
```

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 *RTI Data Distribution Service* also supports the **MATCH** operator as an extended operator (see Section 5.4.6.4).

The following sections provide more information:

- ❏ SQL Grammar (Section 5.4.6.1)
- ❏ Token Expressions (Section 5.4.6.2)
- ❏ Type Compatibility in the Predicate (Section 5.4.6.3)
- ❏ SQL Extension: Regular Expression Matching (Section 5.4.6.4)
- ❏ Composite Members (Section 5.4.6.5)
- ❏ Strings (Section 5.4.6.6)
- ❏ Enumerations (Section 5.4.6.7)
- ❏ Pointers (Section 5.4.6.8)
- ❏ Arrays (Section 5.4.6.9)
- ❏ Sequences (Section 5.4.6.10)

5.4.6.1 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.

```
Expression ::= FilterExpression
           |   TopicExpression
           |   QueryExpression
           .
FilterExpression ::= Condition
TopicExpression  ::= SelectFrom { Where } ';'
QueryExpression  ::= { Condition }{ 'ORDER BY' (FIELDNAME // ',') }
                     .

SelectFrom       ::= 'SELECT' Aggregation 'FROM' Selection
                 .
Aggregation      ::= '*'
                 |   (SubjectFieldSpec // ',')
                 .
SubjectFieldSpec ::= FIELDNAME
                 |   FIELDNAME 'AS' IDENTIFIER
                 |   FIELDNAME IDENTIFIER
                 .
Selection   ::= TOPICNAME
            |   TOPICNAME NaturalJoin JoinItem
            .
JoinItem    ::= TOPICNAME
            |   TOPICNAME NaturalJoin JoinItem
            |   '(' TOPICNAME NaturalJoin JoinItem ')'
            .
NaturalJoin ::= 'INNER JOIN'
            |   'INNER NATURAL JOIN'
            |   'NATURAL JOIN'
            |   'NATURAL INNER JOIN'
            .
Where       ::= 'WHERE' Condition
            .
Condition   ::= Predicate
            |   Condition 'AND' Condition
            |   Condition 'OR' Condition
            |   'NOT' Condition
            |   '(' Condition ')'
```

```
             .
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
           |   PARAMETER
           .
```

Note: INNER JOIN, INNER NATURAL JOIN, NATURAL JOIN, and NATURAL INNER JOIN are all aliases, in the sense that they have the same semantics. They are all supported because they all are part of the SQL standard.

5.4.6.2 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.

```
IDENTIFIER: LETTER (PART_LETTER)*
where LETTER: [ "A"-"Z","_","a"-"z" ]
      PART_LETTER: [ "A"-"Z","_","a"-"z","0"-"9" ]
```

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 either boolean, octet, (unsigned) short, (unsigned) long, (unsigned) long long, float double, char, wchar, string, wstring, or enum.

```
FIELDNAME: FieldNamePart ( "." FieldNamePart )*
where FieldNamePart : IDENTIFIER ( "[" Index "]" )*
      Index> : (["0"-"9"])+
             | ["0x","0X"](["0"-"9", "A"-"F", "a"-"f"])+
```

Primitive IDL types referenced by FIELDNAME are treated as different types in Predicate according to the following table:

Predicate Data Type	IDL Type
BOOLEANVALUE	boolean
INTEGERVALUE	octet, (unsigned) short, (unsigned) long, (unsigned) long long
FLOATVALUE	float, double
CHARVALUE	char, wchar
STRING	string, wstring
ENUMERATEDVALUE	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.

```
TOPICNAME : IDENTIFIER
```

INTEGERVALUE—Any series of digits, optionally preceded by a plus or minus sign, representing a decimal integer value within the range of the system. 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.

```
CHARVALUE : "'" (~["'"])? "'"
```

FLOATVALUE—Any series of digits, optionally preceded by a plus or minus sign and optionally including a floating point ('.'). 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)?
  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"]
```

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 : "%" (["0"-"9"])+
```

5.4.6.3 Type Compatibility in the Predicate

As seen in Table 5.6, only certain combinations of type comparisons are valid in the Predicate.

5.4.6.4 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.

Table 5.6 **Valid Type Comparisons**

	BOOLEAN VALUE	INTEGER VALUE	FLOAT VALUE	CHAR VALUE	STRING	ENUMERATED VALUE
BOOLEAN	YES					
INTEGERVALUE		YES	YES			
FLOATVALUE		YES	YES			
CHARVALUE				YES	YES	YES
STRING				YES	YES [a]	YES
ENUMERATED VALUE		YES		YES [b]	YES [b]	YES [c]

a. See Section 5.4.6.4.

b. Because 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.

c. Only for same-type Enums.

MATCH is case-sensitive. These characters have special meaning: `,/?*[]-^!\%`

The pattern allows limited "wild card" matching under the rules in Table 5.7 on page 5-28.

The syntax is similar to the POSIX® **fnmatch** syntax[1]. The MATCH syntax is also similar to the 'subject' strings of TIBCO Rendezvous®. Some example expressions include:

```
"symbol MATCH 'NASDAQ/[A-G]*'"
"symbol MATCH 'NASDAQ/GOOG,NASDAQ/MSFT'"
```

Table 5.7 **Wild Card Matching**

Character	Meaning
,	A , separates a list of alternate patterns. The field string is matched if it matches one or more of the patterns.
/	A / in the pattern string matches a / in the field string. It separates a sequence of mandatory substrings.

1. See http://www.opengroup.org/onlinepubs/000095399/functions/fnmatch.html.

Table 5.7 **Wild Card Matching**

Character	Meaning
`?`	A `?` in the pattern string matches any single non-special characters in the field string.
`*`	A `*` in the pattern string matches 0 or more non-special characters in field string.
`%`	This special character is used to designate filter expression parameters.
`\`	*(Not supported)* Escape character for special characters.
`[charlist]`	Matches any one of the characters in charlist.
`[!charlist]` or `[^charlist]`	*(Not supported)* Matches any one of the characters *not* in charlist.
`[s-e]`	Matches any character from `s` to `e`, inclusive.
`[!s-e]` or `[^s-e]`	*(Not supported)* Matches any character *not* in the interval `s` to `e`.

5.4.6.5 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 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.

5.4.6.6 Strings

The filter expression and parameters can use IDL strings. String constants must appear between single quotation marks (`'`).

For example:

```
" fish = 'salmon' "
```

5.4.6.7 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'"
```

5.4.6.8 Pointers

Pointers can be used in filter expressions and are automatically dereferenced to the correct value.

For example:

```
struct Point {
    long x;
    long 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"
```

5.4.6.9 Arrays

Arrays are accessed with the familiar `[]` notation.

For example:

```
struct ArrayType {
    long value[255][5];
};
```

The following expression is valid on a *Topic* of type ArrayType:

```
"value[244][2] = 5"
```

5.4.6.10 Sequences

Sequence elements can be accessed using the `()` or `[]` notation.

For example:

```
struct SequenceType {
    sequence<long> s;
};
```

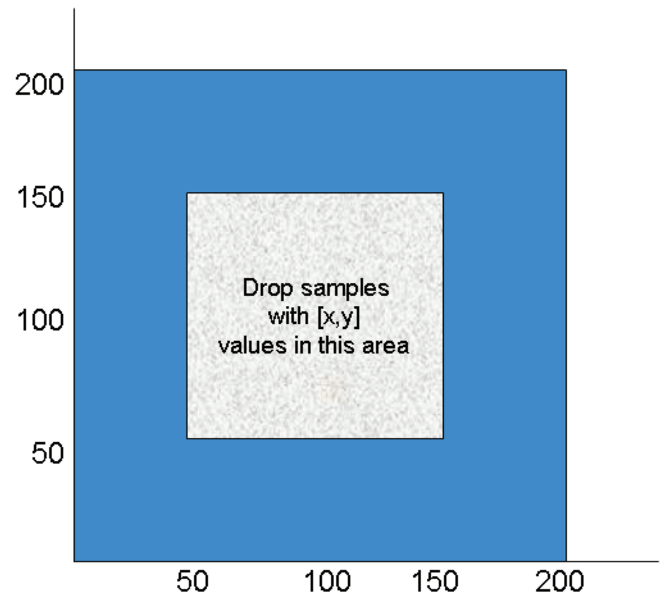
The following expressions are valid on a *Topic* of type SequenceType:

```
"s(1) = 5"
"s[1] = 5"
```

5.4.6.11 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 samples that you are not interested in. Instead you only want to receive samples *outside* the middle of the rectangle, as seen in Figure 5.5. That is, you want to filter *out* data points in the gray box.

Figure 5.5 **Filtering Example**



The filter expression would look like this (remember the expression is written so that samples that we *do* want will *pass*):

```
"(X < 50 or X > 150) and (Y < 50 or Y > 150)"
```

While this filter works, it cannot be changed after the ContentFilteredTopic has been created. 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 using filter parameters. An more flexible way to write the expression is this:

```
"(X < %0 or X > %1) and (Y < %2 or Y > %3)"
```

Recall that when you create a ContentFilteredTopic (see Section 5.4.3), 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 online documentation (from the **Modules** page, select **Infrastructure**).

In C++, the filter parameters could be assigned like this:

```
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:

```
FilterParameter[0] = 10
FilterParameter[1] = 190
set_expression_parameters(....)
```

Note: The new filter parameters will affect all *DataReaders* that have been created with this ContentFilteredTopic.

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* MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12) in Market Data applications.

A STRINGMATCH filter expression has the following syntax:

```
<field name> MATCH <string pattern>
```

The relational operator **MATCH** may only be used with string fields. The right-hand operator is a constant string pattern or the filter parameter **%0**. A string pattern specifies a template that the left-hand string (subject string) must match. See Section 5.4.6.4 for a description of the string pattern format.

5.4.7.1 Example STRINGMATCH Filter Expressions

- ❑ 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 'NASDAQ/IBM, NASDAQ/MSFT'
  ```

- ❑ This expression evaluates to TRUE if the value of **symbol** corresponds to **NASDAQ** *and* starts with a letter between M and Y:

  ```
  symbol MATCH 'NASDAQ/[M-Y]*'
  ```

5.4.7.2 STRINGMATCH Filter Expression Parameters

Filter expression parameters are provided to the ContentFilteredTopic as a string sequence (see Section 5.4.3 and Section 5.4.5).

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.

In the builtin SQL filter, parameters have the syntax `%<parameter index>` and must be specified explicitly. For example:

```
position.x < %0 and position.y > %1
```

In the builtin STRINGMATCH filter, there is always a parameter, the parameter 0. The parameter can be specified explicitly using the same syntax of the SQL filter or implicitly using a constant string pattern. For example:

`symbol MATCH %0` (Explicit parameter)
`symbol MATCH IBM` (Implicit parameter initialized to IBM)

5.4.8 Custom Content Filters

By default, a ContentFilteredTopic will use a SQL-like content filter, DDS_SQLFILTER_NAME[1] (see SQL Filter Expression Notation (Section 5.4.6)), which implements a superset of the DDS-specified content filter. There is another builtin filter, **DDS_STRINGMATCHFILTER_NAME** (see STRINGMATCH Filter Expression Notation (Section 5.4.7)). 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 Creating ContentFilteredTopics (Section 5.4.3)).

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.

Notes:

- ❑ The API for using a custom content filter is subject to change in a future release.
- ❑ Custom content filters are not supported when using the .NET APIs.

5.4.8.1 Registering a Custom Filter

To register a new filter, use the *DomainParticipant's* **register_contentfilter()** operation[2]:

```
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.

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**, **evaluate** and **finalize** functions described below. 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. *RTI Data Distribution Service* will call this function when a ContentFilteredTopic is created and when the filter parameters are changed. This parameter cannot be NULL. See Section 5.4.8.4 for details.

1. See the DDS specification for details.
2. This operation is an extension to the DDS standard.

- ❑ **evaluate** The function that will be called by *RTI Data Distribution Service* each time a sample is received. Its purpose is to evaluate the sample based on the filter. This parameter cannot be NULL. See Section 5.4.8.5 for details.
- ❑ **finalize** The function that will be called by *RTI Data Distribution Service* when an instance of the custom content filter is no longer needed. This parameter *may* be NULL. See Section 5.4.8.6 for details.

5.4.8.2 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.)

```
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**.

If there are still existing discovered *DataReaders* with the same **filter_name** and the filter's **compile()** method has previously been called on the discovered *DataReaders*, the filter's **finalize()** method 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*.

5.4.8.3 Retrieving a ContentFilter

If you know the name of a ContentFilter, you can get a pointer to its structure. If the ContentFilter has been already been registered, this operation will return NULL.

```
DDS_ContentFilter *lookup_contentfilter (const char * filter_name)
```

5.4.8.4 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.

1. This operation is an extension to the DDS standard.

5. Topics

The only requirement is that the **error_code** parameter passed to the compile function must return **OK** on successful execution. For example:

```
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;
}
```

new_compile_data A user-specified opaque pointer of this instance of the content filter. This value is passed to **evaluate()** and **finalize()**.

expression An ASCIIZ string with the filter expression the ContentFilteredTopic was created with. Note that the memory used by the parameter pointer is owned by *RTI Data Distribution Service*. 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 Section 5.4.3).

Important: The sequence passed to the compile function is owned by *RTI Data Distribution Service* 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 Using Generated Types without RTI Data Distribution Service (Standalone) (Section 3.7). [Note: if you are using the Java API, this parameter will always be NULL.]

type_class_name Fully qualified class name of the related *Topic*.

old_compile_data The **new_compile_data** value from a *previous* call to this instance of a content filter. If **compile()** is called more than once for an instance of a ContentFilteredTopic (such as if the expression parameters are changed), then the **new_compile_data** value returned by the previous invocation is passed in the

old_compile_data parameter (which can be NULL). If this is a new instance of the filter, NULL is passed. This parameter is useful for freeing or reusing previously allocated resources.

5.4.8.5 Evaluate Function

The **evaluate** function specified in the ContentFilter will be called each time a sample is received. This function's purpose is to determine if a sample should be filtered out (not put in the receive queue).

For example:

```
DDS_Boolean sample_evaluate_function(
                  void* compile_data, const void* sample) {
    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.

5.4.8.6 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:

```
void sample_finalize_function ( void* compile_data) {

    /* free parameter string from compile function */
    DDS_String_free((char *)compile_data);
}
```

The function may use the following optional parameters:

system_key See Section 5.4.8.4.

handle This is the opaque returned by the last call to the compile function.

Chapter 6 Sending Data

This chapter discusses how to create, configure, and use *Publishers* and *DataWriter*s to send data. It describes how these entities interact, as well as the types of operations that are available for them.

This chapter includes the following sections:

- ❏ Preview: Steps to Sending Data (Section 6.1)
- ❏ Publishers (Section 6.2)
- ❏ DataWriters (Section 6.3)
- ❏ Publisher/Subscriber QosPolicies (Section 6.4)
- ❏ DataWriter QosPolicies (Section 6.5)
- ❏ Flow Controllers (DDS Extension) (Section 6.6)

The goal of this chapter 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 online documentation.

6.1 Preview: Steps to Sending Data

To send samples of a data instance:

1. Create and configure the required Entities:
   a. Create a *DomainParticipant*.
   b. Register user data types[1] with the *DomainParticipant*. For example, the '**FooDataType**'.

c. Use the *DomainParticipant* to create a *Topic* with the registered data type.

d. Optionally[1], use the *DomainParticipant* to create a *Publisher*.

e. Use the *Publisher* or *DomainParticipant* to create a *DataWriter* for the *Topic*.

f. Use a type-safe method to cast the generic *DataWriter* created by the *Publisher* to a type-specific *DataWriter*. For example, '***FooDataWriter***'.

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 Section 2.2.2 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, you should 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 *RTI Data Distribution Service* internal buffers from where the data sample is sent under the criteria set by the *Publisher's* and *DataWriter's* QosPolicies. If there are matched *DataReaders*, then the data sample will have been passed to the physical transport plug-in/device driver by the time that **write()** returns.

1. Type registration is not required for built-in types (see Section 3.2.1).

1. You are not required to explicitly create a *Publisher*; instead, you can use the 'implicit *Publisher*' created from the *DomainParticipant*. See Creating Publishers Explicitly vs. Implicitly (Section 6.2.1).

6.2 Publishers

An application that intends to publish information needs the following DDS Entities: *DomainParticipant*, *Topic*, *Publisher*, and *DataWriter*. All DDS Entities have a corresponding specialized *Listener* and a set of QosPolicies. A *Listener* is how *RTI Data Distribution Service* notifies your application of status changes relevant to the DDS Entity. The QosPolicies allow your application to configure the behavior and resources of the DDS Entity.

- A *DomainParticipant* defines the 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 QosPolicies 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 Creating Publishers Explicitly vs. Implicitly (Section 6.2.1).

Figure 6.1 on page 6-4 shows how these *Entities* are related, as well as the methods defined for each *Entity*.

Publishers are used to perform the operations listed in Table 6.1 on page 6-5. 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 online documentation.

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

Figure 6.1 **DDS Publication Module**

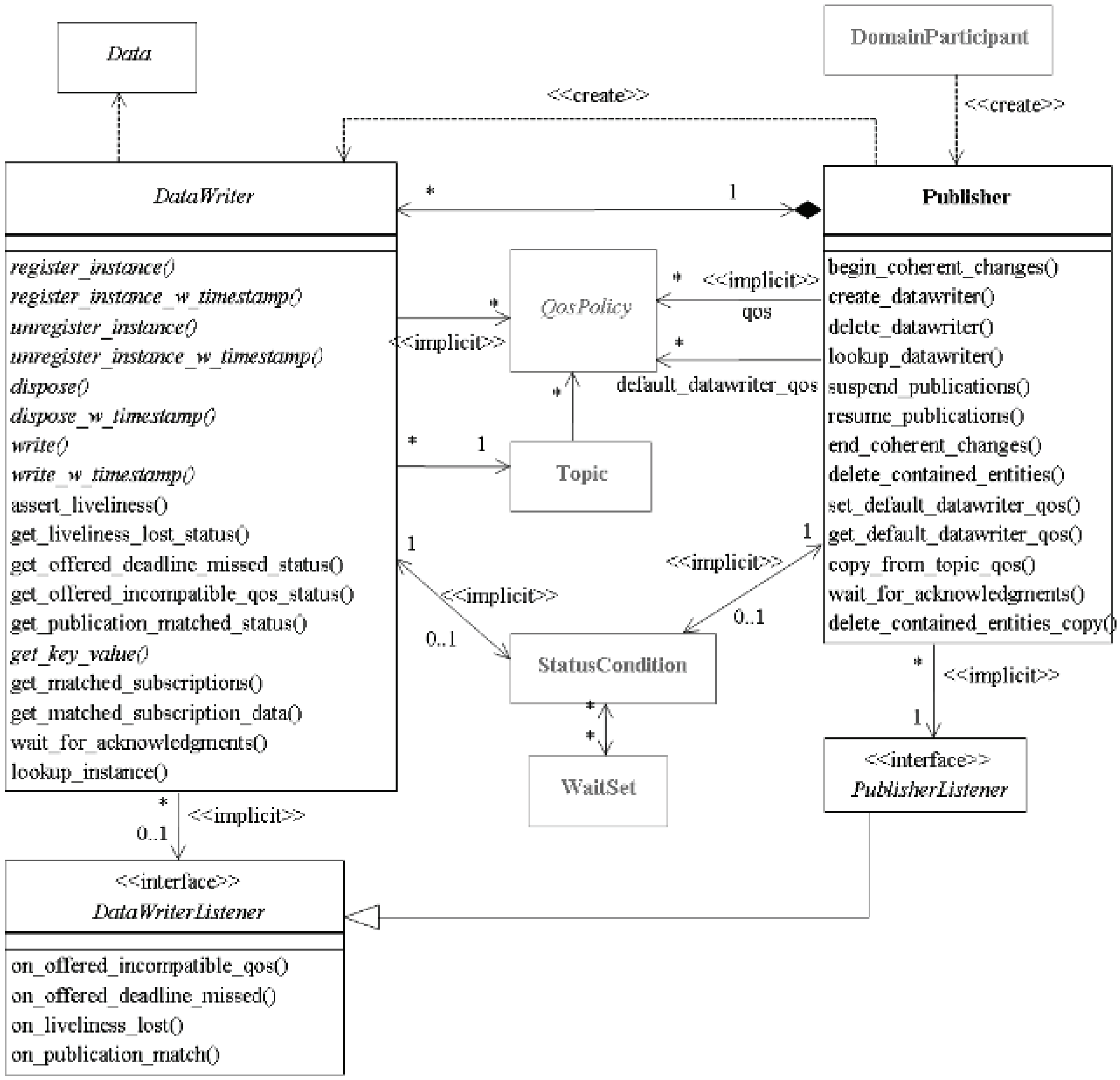


Table 6.1 **Publisher Operations**

Working with ...	Operation	Description	Reference
DataWriters	begin_coherent_changes	Indicates that the application will begin a coherent set of modifications.	Section 6.3.9
	create_datawriter	Creates a *DataWriter* that will belong to the Publisher.	Section 6.3.1
	create_datawriter_with_profile	Sets the *DataWriter's* QoS based on a specified QoS profile.	
	copy_from_topic_qos	Copies relevant QosPolicies from a *Topic* into a DataWriterQoS structure.	Section 6.2.4.5
	delete_contained_entities	Deletes all of the *DataWriters* that were created by the *Publisher*.	Section 6.2.3.1
	delete_datawriter	*Deletes* a *DataWriter* that belongs to the *Publisher*.	Section 6.3.2
	end_coherent_changes	Ends the coherent set initiated by begin_coherent_changes().	Section 6.3.9
	get_default_datawriter_qos	Copies the *Publisher's* default DataWriterQoS values into a DataWriterQos structure.	Section 6.3.12
	get_status_changes	Will always return 0 since there are no Statuses currently defined for *Publishers*.	Section 4.1.4
	lookup_datawriter	Retrieves a *DataWriter* previously created for a specific *Topic*.	Section 6.2.6
	set_default_datawriter_qos	Sets or changes the default DataWriterQos values.	Section 6.2.4.4
	set_default_datawriter_qos_with_profile	Sets or changes the default DataWriterQos values based on a QoS profile.	
Libraries and Profiles	get_default_library	Gets the *Publisher's* default QoS profile library.	Section 6.2.4.3
	get_default_profile	Gets the *Publisher's* default QoS profile.	
	get_default_profile_library	Gets the library that contains the *Publisher's* default QoS profile.	
	set_default_library	Sets the default library for a *Publisher*.	
	set_default_profile	Sets the default profile for a *Publisher*.	
Participants	get_participant	Gets the *DomainParticipant* that was used to create the *Publisher*.	Section 6.2.6

Table 6.1 **Publisher Operations**

Working with ...	Operation	Description	Reference
Publishers	enable	Enables the *Publisher*.	Section 4.1.2
	get_qos	Gets the *Publisher's* current QosPolicy settings. This is most often used in preparation for calling set_qos().	Section 6.2.4
	set_qos	Sets the *Publisher's* QoS. You can use this operation to change the values for the *Publisher's* QosPolicies. Note, however, that not all QosPolicies can be changed after the *Publisher* has been created.	
	set_qos_with_profile	Sets the *Publisher's* QoS based on a specified QoS profile.	
	get_listener	Gets the currently installed Listener.	Section 6.2.5
	set_listener	Sets the *Publisher's* Listener. If you created the *Publisher* without a Listener, you can use this operation to add one later.	
	suspend_publications	Provides a *hint* that multiple data-objects within the Publisher are about to be written. *RTI Data Distribution Service* does not currently use this hint.	Section 6.2.9
	resume_publications	Reverses the action of suspend_publications().	

6.2.1 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 Section 8.3.9). You can use this 'implicit *Pub-*

lisher' just like any other *Publisher* (it has the same operations, QosPolicies, 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.

6.2.2 Creating Publishers

Before you can explicitly create a *Publisher*, you need a *DomainParticipant* (see Section 8.3). To create a *Publisher*, use the *DomainParticipant's* **create_publisher()** or **create_publisher_with_profile()** operations:

```
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)
```

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 Chapter 15: Configuring QoS with XML.

qos If you want the default QoS settings (described in the online documentation), use **DDS_PUBLISHER_QOS_DEFAULT** for this parameter (see Figure 6.2). If you want to customize any of the QosPolicies, supply a QoS structure (see Figure 6.3). The QoS structure for a *Publisher* is described in Section 6.4.

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. *RTI Data Distribution Service* 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 Section 6.2.5.

mask 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 Listeners (Section 4.4).

library_name A QoS Library is a named set of QoS profiles. See QoS Libraries (Section 15.9). If NULL is used for **library_name**, the *DomainParticipant's* default library is assumed (see Section 6.2.4.3).

profile_name A QoS profile groups a set of related QoS, usually one per entity. See QoS Profiles (Section 15.8). If NULL is used for **profile_name**, the *DomainParticipant's* default profile is assumed and **library_name** is ignored.

Figure 6.2 **Creating a Publisher with Default QosPolicies**

```
// create the publisher
DDSPublisher* publisher =
    participant->create_publisher(DDS_PUBLISHER_QOS_DEFAULT,
                                  NULL,
                                  DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
      // handle error
};
```

For more examples, see Configuring QoS Settings when the Publisher is Created (Section 6.2.4.1).

After you create a *Publisher*, the next step is to use the *Publisher* to create a *DataWriter* for each *Topic*, see Section 6.3.1. For a list of operations you can perform with a *Publisher*, see Table 6.1 on page 6-5.

6.2.3 Deleting Publishers

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 (Section 6.2.3.1) 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 Restricted Operations in Listener Callbacks (Section 4.5.1).

6.2.3.1 Deleting Contained DataWriters

The *Publisher's* **delete_contained_entities()** operation deletes all the *DataWriters* that were created by the *Publisher*.

```
DDS_ReturnCode_t delete_contained_entities ()
```

After this operation returns successfully, the application may delete the *Publisher* (see Section 6.2.3).

6.2.4 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:

```
DDS_PublisherQos struct  {
    DDS_PresentationQosPolicy           presentation;
    DDS_PartitionQosPolicy              partition;
    DDS_GroupDataQosPolicy              group_data;
    DDS_EntityFactoryQosPolicy          entity_factory;
    DDS_AsynchronousPublisherQosPolicy asynchronous_publisher;
    DDS_ExclusiveAreaQosPolicy          exclusive_area;
} DDS_PublisherQos;
```

6. Sending Data

Note: set_qos() cannot always be used in a listener callback; see Restricted Operations in Listener Callbacks (Section 4.5.1).

Table 6.2 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 online documentation for each policy.

Table 6.2 **Publisher QosPolicies**

QosPolicy	Description
ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1)	Configures the mechanism that sends user data in an external middleware thread.
ENTITYFACTORY QosPolicy (Section 6.4.2)	Controls whether or not child entities are created in the enabled state.
EXCLUSIVE_AREA QosPolicy (DDS Extension) (Section 6.4.3)	Configures multi-thread concurrency and deadlock prevention capabilities.
GROUP_DATA QosPolicy (Section 6.4.4)	Along with TOPIC_DATA QosPolicy (Section 5.2.1) and USER_DATA QosPolicy (Section 6.5.23), this QosPolicy is used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data.
PARTITION QosPolicy (Section 6.4.5)	Adds string identifiers that are used for matching *DataReaders* and *DataWriters* for the same *Topic*.
PRESENTATION QosPolicy (Section 6.4.6)	Controls how DDS presents data received by an application to the *DataReaders* of the data.

6.2.4.1 Configuring QoS Settings when the Publisher is Created

As described in Creating Publishers (Section 6.2.2), 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 6.2 on page 6-8 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 Section 8.3.6.4).

- To create a *Publisher* with non-default QoS settings, without using a QoS profile, see Figure 6.3 on page 6-11. 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 6.4 on page 6-12.
- 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 6.5 on page 6-12.

For more information, see Creating Publishers (Section 6.2.2) and Chapter 15: Configuring QoS with XML.

Figure 6.3 **Creating a Publisher with Non-default QosPolicies (not from a profile)**

```
DDS_PublisherQos publisher_qos;[1]

// 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
}
```

1. For the C API, you need to use **DDS_PublisherQos_INITIALIZER** or **DDS_PublisherQos_initialize()**. See Special QosPolicy Handling Considerations for C (Section 4.2.2)

Figure 6.4 **Creating a Publisher with a QoS Profile**

```
// create the publisher with QoS profile
DDSPublisher* publisher =
           participant->create_publisher_with_profile(
                                         "MyPublisherLibary",
                                         "MyPublisherProfile",
                                         NULL, DDS_STATUS_MASK_NONE);
if (publisher == NULL) {
    // handle error
}
```

Figure 6.5 **Getting QoS Values from a Profile, Changing QoS Values, Creating a Publisher with Modified QoS Values**

```
DDS_PublisherQos publisher_qos;[1]

// 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 Special QosPolicy Handling Considerations for C (Section 4.2.2)

6.2.4.2 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 6.6. 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 QosPolicies cannot be changed after the *Publisher* has been enabled—this restriction is noted in the descriptions of the individual QosPolicies.
- 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 6.7. For more information, see Chapter 15: Configuring QoS with XML.

Figure 6.6 **Changing the Qos of an Existing Publisher**

```
DDS_PublisherQos publisher_qos;[1]
// 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
}
```

1. For the C API, you need to use **DDS_PublisherQos_INITIALIZER** or **DDS_PublisherQos_Initialize()**. See Special QosPolicy Handling Considerations for C (Section 4.2.2)

Figure 6.7 **Changing the QoS of an Existing Publisher with a QoS Profile**

```
retcode = publisher->set_qos_with_profile(
                  "PublisherProfileLibrary","PublisherProfile");

if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

6.2.4.3 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 Chapter 15: Configuring QoS with XML.

```
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:

```
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 Section 6.2.4.4 (you may pass in NULL after having called 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 Section 8.3.6.4.

6.2.4.4 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:

```
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 *DataWriters* 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 else, 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(),**

6.2.4.5 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 Section 5.1.3, 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 Section 6.3.12).

- **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 QosPolicies 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.

6.2.5 Setting Up PublisherListeners

Like all DDS 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 *RTI Data Distribution Service* 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 Section 4.4.

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

As illustrated in Figure 6.1 on page 6-4, 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 Section 6.3.3. If a *DataWriterListener* has been installed and has been enabled to handle a *Status* change for the *DataWriter*, then *RTI Data Distribution Service* 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,

```
DDS_StatusMask mask = DDS_OFFERED_DEADLINE_MISSED_STATUS |
                     DDS_OFFERED_INCOMPATIBLE_QOS_STATUS;
publisher = participant->create_publisher(DDS_PUBLISHER_QOS_DEFAULT,
                                          listener, mask);
```

or

```
DDS_StatusMask mask = DDS_OFFERED_DEADLINE_MISSED_STATUS |
                      DDS_OFFERED_INCOMPATIBLE_QOS_STATUS;
publisher->set_listener(listener, mask);
```

As previously mentioned, the callbacks in the *PublisherListener* act as 'default' callbacks for all the *DataWriter*s contained within. When *RTI Data Distribution Service* 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, *RTI Data Distribution Service* dispatches the event to the *DataWriterListener* callback. Otherwise, *RTI Data Distribution Service* 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 Section 6.3.3.

Similarly, the callbacks in the *DomainParticipantListener* act as 'default' callbacks for all the *Publishers* that belong to it. For more information on *DomainParticipantListeners*, see Section 8.3.5.

For example, Figure 6.8 shows how to create a *Publisher* with a *Listener* that simply prints the events it receives.

6.2.6 Finding a Publisher's Related 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.
- ❏ **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:

  ```
  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.

6.2.7 Waiting for Acknowledgments

The *Publisher's* **wait_for_acknowledgments()** operation blocks the calling thread until either all data written by the Publisher's reliable *DataWriters* is acknowledged by all matched reliable *DataReaders*, or else the duration specified by the **max_wait** parameter elapses, whichever happens first.

```
DDS_ReturnCode_t wait_for_acknowledgments
                                        (const DDS_Duration_t &  max_wait)
```

This operation returns **DDS_RETCODE_OK** if all the samples were acknowledged, or **DDS_RETCODE_TIMEOUT** if the **max_wait** duration expired first.

There is a similar operation available for individual *DataWriters*, see Section 6.3.10.

The reliability protocol used by *RTI Data Distribution Service* is discussed in Chapter 10: Reliable Communications.

Figure 6.8 **Example Code to Create a Publisher with a Simple Listener**

```
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\n");
}

// ...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);
```

6. Sending Data

6.2.8 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*.

- ❏ OFFERED_DEADLINE_MISSED Status (Section 6.3.5.4)
- ❏ LIVELINESS_LOST Status (Section 6.3.5.3)
- ❏ OFFERED_INCOMPATIBLE_QOS Status (Section 6.3.5.5)
- ❏ PUBLICATION_MATCHED Status (Section 6.3.5.6)
- ❏ RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7)
- ❏ RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.5.8)

6.2.9 Suspending and Resuming Publications

The operations **suspend_publications()** and **resume_publications()** provide a *hint* to *RTI Data Distribution Service* that multiple data-objects within the Publisher are about to be written. *RTI Data Distribution Service* does not currently use this hint.

6.3 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 6.3. The most important operation is **write()**, described in Section 6.3.7. For more details on all operations, see the online documentation.

DataWriters are created by using operations on a *DomainParticipant* or a *Publisher*, as described in Section 6.3.1. 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 Restricted Operations in Listener Callbacks (Section 4.5.1).

Table 6.3 **DataWriter Operations**

Working with ...	Operation	Description	Reference
Data-Writers	assert_liveliness	Manually asserts the liveliness of the *DataWriter*.	Section 6.3.14
	enable	Enables the *DataWriter*.	Section 4.1.2
	get_qos	Gets the QoS.	Section 6.3.12
	lookup_instance	Gets a handle, given an instance. (Useful for keyed data types only.)	Section 6.3.11.3
	set_qos	Modifies the QoS.	Section 6.3.12
	set_qos_with_profile	Modifies the QoS based on a QoS profile.	Section 6.3.12
	get_listener	Gets the currently installed Listener.	Section 6.3.3
	set_listener	Replaces the Listener.	
FooData-Writer (See Section 6.3.6)	dispose	States that the instance no longer exists. (Useful for keyed data types only.)	Section 6.3.11.2
	dispose_w_timestamp	Same as dispose, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)	
	flush	Makes the batch available to be sent on the network.	Section 6.3.8
	get_key_value	Maps an instance_handle to the corresponding key.	Section 6.3.11.4
	narrow	A type-safe way to cast a pointer. This takes a DDS-DataWriter pointer and 'narrows' it to a 'Foo-DataWriter' where 'Foo' is the related data type.	Section 6.3.6
	register_instance	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.)	Section 6.3.11.1
	register_instance_w_timestamp	Like register_instance, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)	
	unregister_instance	Reverses register_instance. Relinquishes the ownership of the instance. (Useful for keyed data types only.)	
	unregister_instance_w_timestamp	Like unregister_instance, but allows the application to override the automatic source_timestamp. (Useful for keyed data types only.)	
	write	Writes a new value for a data-instance.	Section 6.3.7
	write_w_timestamp	Same as write, but allows the application to override the automatic source_timestamp.	

Table 6.3 **DataWriter Operations**

Working with ...	Operation	Description	Reference
Matched Subscriptions	get_matched_subscriptions	Gets a list of subscriptions that have a matching *Topic* and compatible QoS. These are the subscriptions currently associated with the *DataWriter*.	Section 6.3.13.1
	get_matched_subscription_data	Gets information on a subscription with a matching *Topic* and compatible QoS.	
	get_matched_subscription_locators	Gets a list of locators for subscriptions that have a matching Topic and compatible QoS. These are the subscriptions currently associated with the *DataWriter*.	
Other	get_publisher	Gets the *Publisher* to which the *DataWriter* belongs.	Section 6.3.13.2
	get_topic	Get the Topic associated with the *DataWriter*.	
	wait_for_acknowledgements	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.	Section 6.3.10
Status	get_status_changes	Gets a list of statuses that have changed since the last time the application read the status or the listeners were called.	Section 4.1.4
	get_liveliness_lost_status	Gets LIVELINESS_LOST status.	Section 6.3.5
	get_offered_deadline_missed_status	Gets OFFERED_DEADLINE_MISSED status.	
	get_offered_incompatible_qos_status	Gets OFFERED_INCOMPATIBLE_QOS status.	
	get_publication_match_status	Gets PUBLICATION_MATCHED_QOS status.	
	get_reliable_writer_cache_changed_status	Gets RELIABLE_WRITER_CACHE_CHANGED status	
	get_reliable_reader_activity_changed_status	Gets RELIABLE_READER_ACTIVITY_CHANGED status	
	get_datawriter_cache_status	Gets DATA_WRITER_CACHE_status	
	get_datawriter_protocol_status	Gets DATA_WRITER_PROTOCOL status	

Table 6.3 **DataWriter Operations**

Working with ...	Operation	Description	Reference
Status (cont'd)	get_matched_subscription_datawriter_protocol_status	Gets DATA_WRITER_PROTOCOL status for this *DataWriter*, per matched subscription identified by the subscription_handle.	Section 6.3.5
	get_matched_subscription_datawriter_protocol_status_by_locator	Gets DATA_WRITER_PROTOCOL status for this *DataWriter*, per matched subscription as identified by a locator.	

6.3.1 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 Section 6.2.1. If you use a *Publisher's* operations to create a *DataWriter*, it will belong to that *Publisher.*

```
DDSDataWriter* create_datawriter ( DDSTopic *topic,
                                   const DDS_DataWriterQos &qos,
                                   DDSDataWriterListener *listener,
                                   DDS_StatusMask mask)

DDSDataWriter * create_datawriter_with_profile (
                                   DDSTopic * topic,
                                   const char * library_name,
                                   const char * profile_name,
                                   DDSDataWriterListener * listener,
                                   DDS_StatusMask mask)
```

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 Chapter 15: Configuring QoS with XML.

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 online documentation), use the constant **DDS_DATAWRITER_QOS_DEFAULT** for this parameter (see Figure 6.9). If you want to customize any of the QosPolicies, supply a QoS structure (see Section 6.3.12).

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. *RTI Data Distribution Service* 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 Section 6.3.3.

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 Listeners (Section 4.4).

library_name A QoS Library is a named set of QoS profiles. See QoS Libraries (Section 15.9).

profile_name A QoS profile groups a set of related QoS, usually one per entity. See QoS Profiles (Section 15.8).

Figure 6.9 **Creating a DataWriter with Default QosPolicies and a Listener**

```
// 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);
```

For more examples on how to create a *DataWriter*, see Configuring QoS Settings when the DataWriter is Created (Section 6.3.12.1)

After you create a *DataWriter*, you can use it to write data. See Section 6.3.7.

Note: When a *DataWriter* is created, only those transports already registered are available to the *DataWriter.* The builtin transports are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataWriter* is created, or (c) you look up a builtin data reader, whichever happens first.

6.3.2 Deleting DataWriters

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 Restricted Operations in Listener Callbacks (Section 4.5.1)

To delete all of a *Publisher's DataWriters*, use the *Publisher's* **delete_contained_entities()** operation (see Section 6.2.3.1).

Special instructions for deleting *DataWriters* if you are using the 'Timestamp' APIs and BY_SOURCE_TIMESTAMP Destination Order:

This note only applies when the *DataWriter's* DestinationOrderQosPolicy's **kind** is BY_SOURCE_TIMESTAMP.

Calls to **delete_datawriter()** may fail if your application has previously used the "with timestamp" APIs (**write_w_timestamp()**, **register_instance_w_timestamp()**, **unregister_instance_w_timestamp()**, or **dispose_w_timestamp()**) with a timestamp that is larger than the time at which **delete_datawriter()** is called.

To prevent **delete_datawriter()** from failing in this situation, either:

❏ Change the WriterDataLifeCycle QoS Policy so that *RTI Data Distribution Service* will not autodispose unregistered instances:

```
writer_qos.writer_data_lifecycle.
autodispose_unregistered_instances =
       DDS_BOOLEAN_FALSE;
```

or

❏ Explicitly call **unregister_instance_w_timestamp()** for all instances modified with the ***_w_timestamp()** APIs before calling **delete_datawriter()**.

6.3.3 Setting Up DataWriterListeners

*DataWriter*s may optionally have *Listeners*. *Listeners* are essentially callback routines and provide the means for *RTI Data Distribution Service* to notify your application of the occurrence of events (status changes) relevant to the *DataWriter*. For more general information on *Listeners*, see Listeners (Section 4.4).

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

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 Section 6.2.5 and Section 8.3.5).

Listeners are typically set up when the *DataWriter* is created (see Section 6.2). You can also set one up after creation by using the **set_listener()** operation. *RTI Data Distribution Service* will invoke a *DataWriter*'s *Listener* to report the status changes listed in Table 6.4 (if the Listener is set up to handle the particular status, see Section 6.3.3).

Table 6.4 **DataWriterListener Callbacks**

This DataWriterListener callback...	... is triggered by ...
on_instance_replaced()	A replacement of an existing instance by a new instance; see Configuring DataWriter Instance Replacement (Section 6.5.18.2)
on_liveliness_lost	A change to LIVELINESS_LOST Status (Section 6.3.5.3)
on_offered_deadline_missed	A change to OFFERED_DEADLINE_MISSED Status (Section 6.3.5.4)
on_offered_incompatible_qos	A change to OFFERED_INCOMPATIBLE_QOS Status (Section 6.3.5.5)
on_publication_matched	A change to PUBLICATION_MATCHED Status (Section 6.3.5.6)
on_reliable_writer_cache_changed	A change to RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7)
on_reliable_reader_activity_changed	A change to RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.5.8)

6. Sending Data

6.3.4 Checking DataWriter Status

You can access an individual communication status for a *DataWriter* with the operations shown in Table 6.5.

Table 6.5 **DataWriter Status Operations**

Use this operation...	...to retrieve this status:
get_datawriter_cache_status	DATA_WRITER_CACHE_STATUS (Section 6.3.5.1)
get_datawriter_protocol_status	DATA_WRITER_PROTOCOL_STATUS (Section 6.3.5.2)
get_matched_subscription_datawriter_protocol_status	
get_matched_subscription_datawriter_protocol_status_by_locator	
get_liveliness_lost_status	LIVELINESS_LOST Status (Section 6.3.5.3)
get_offered_deadline_missed_status	OFFERED_DEADLINE_MISSED Status (Section 6.3.5.4)
get_offered_incompatible_qos_status	OFFERED_INCOMPATIBLE_QOS Status (Section 6.3.5.5)
get_publication_match_status	PUBLICATION_MATCHED Status (Section 6.3.5.6)
get_reliable_writer_cache_changed_status	RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7)
get_reliable_reader_activity_changed_status	RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.5.8)
get_status_changes	A list of what changed in all of the above.

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 *RTI Data Distribution Service* returns from calling the *Listener* callback associated with that status.

For more on status, see Setting Up DataWriterListeners (Section 6.3.3), Statuses for DataWriters (Section 6.3.5), and Listeners (Section 4.4).

6.3.5 Statuses for DataWriters

There are several types of statuses available for a *DataWriter*. You can use the **get_*_status()** operations (Section 6.3.12) to access them, or use a *DataWriterListener* (Section 6.3.3) to listen for changes in their values. Each status has an associated data structure and is described in more detail in the following sections.

- ❑ DATA_WRITER_CACHE_STATUS (Section 6.3.5.1)
- ❑ DATA_WRITER_PROTOCOL_STATUS (Section 6.3.5.2)
- ❑ LIVELINESS_LOST Status (Section 6.3.5.3)
- ❑ OFFERED_DEADLINE_MISSED Status (Section 6.3.5.4)
- ❑ OFFERED_INCOMPATIBLE_QOS Status (Section 6.3.5.5)
- ❑ PUBLICATION_MATCHED Status (Section 6.3.5.6)
- ❑ RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7)
- ❑ RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.5.8)

6.3.5.1 DATA_WRITER_CACHE_STATUS

This status keeps track of the number of samples in the *DataWriter's* queue.

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 6.6.

Table 6.6 **DDS_DataWriterCacheStatus**

Type	Field Name	Description
DDS_Long	sample_count_peak	Highest number of samples in the *DataWriter's* queue over the lifetime of the *DataWriter*.
DDS_Long	sample_count	Current number of samples in the *DataWriter's* queue.

6.3.5.2 DATA_WRITER_PROTOCOL_STATUS

This status includes internal protocol related metrics (such as the number of samples pushed, pulled, filtered) and the status of wire-protocol traffic.

- *Pulled* samples are samples sent for repairs (that is, samples that had to be resent), for late joiners, and all samples sent by the local *DataWriter* when **push_on_write** (in DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)) is DDS_BOOLEAN_FALSE.
- *Pushed* samples are samples sent on **write()** when **push_on_write** is DDS_BOOLEAN_TRUE.
- *Filtered* samples are samples that are not sent due to *DataWriter* filtering (time-based filtering and ContentFilteredTopics).

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 6.7 on page 6-31):

- **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 Locator Format (Section 12.2.1.1).)

Note: Status for a remote entity is only kept while the entity is alive. Once a remote entity is no longer alive, its status is deleted. If you try to get the matched subscription status for a remote entity that is no longer alive, the 'get status' call will return an error.

Table 6.7 **DDS_DataWriterProtocolStatus**

Type	Field Name	Description
DDS_LongLong	pushed_sample_count	The number of user samples pushed on write from a local *DataWriter* to a matching remote *DataReader*.
	pushed_sample_count_change	The incremental change in the number of user samples pushed on write from a local *DataWriter* to a matching remote *DataReader* since the last time the status was read.
	pushed_sample_bytes	The number of bytes of user samples pushed on write from a local *DataWriter* to a matching remote *DataReader*.
	pushed_sample_bytes_change	The incremental change in the number of bytes of user samples pushed on write from a local *DataWriter* to a matching remote *DataReader* since the last time the status was read.
DDS_LongLong	filtered_sample_count	The number of user samples preemptively filtered by a local *DataWriter* due to Content-Filtered Topics.
	filtered_sample_count_change	The incremental change in the number of user samples preemptively filtered by a local DataWriter due to ContentFilteredTopics since the last time the status was read.
	filtered_sample_bytes	The number of user samples preemptively filtered by a local *DataWriter* due to ContentFiltered-Topics.
	filtered_sample_bytes_change	The incremental change in the number of user samples preemptively filtered by a local *DataWriter* due to ContentFilteredTopics since the last time the status was read.

6. Sending Data

Table 6.7 **DDS_DataWriterProtocolStatus**

Type	Field Name	Description
DDS_LongLong	sent_heartbeat_count	The number of Heartbeats sent between a local *DataWriter* and matching remote *DataReader*s.
	sent_heartbeat_count_change	The incremental change in the number of Heartbeats sent between a local *DataWriter* and matching remote *DataReaders* since the last time the status was read.
	sent_heartbeat_bytes	The number of bytes of Heartbeats sent between a local *DataWriter* and matching remote *DataReader*.
	sent_heartbeat_bytes_change	The incremental change in the number of bytes of Heartbeats sent between a local *DataWriter* and matching remote *DataReaders* since the last time the status was read.
DDS_LongLong	pulled_sample_count	The number of user samples pulled from local *DataWriter* by matching *DataReaders*.
	pulled_sample_count_change	The incremental change in the number of user samples pulled from local *DataWriter* by matching *DataReaders* since the last time the status was read.
	pulled_sample_bytes	The number of bytes of user samples pulled from local *DataWriter* by matching *DataReaders*.
	pulled_sample_bytes_change	The incremental change in the number of bytes of user samples pulled from local *DataWriter* by matching *DataReaders* since the last time the status was read.

Table 6.7 **DDS_DataWriterProtocolStatus**

Type	Field Name	Description
DDS_LongLong	received_ack_count	The number of ACKs from a remote *DataReader* received by a local *DataWriter*.
	received_ack_count_change	The incremental change in the number of ACKs from a remote *DataReader* received by a local *DataWriter* since the last time the status was read.
	received_ack_bytes	The number of bytes of ACKs from a remote DataReader received by a local *DataWriter*.
	received_ack_bytes_change	The incremental change in the number of bytes of ACKs from a remote *DataReader* received by a local *DataWriter* since the last time the status was read.
DDS_LongLong	received_nack_count	The number of NACKs from a remote *DataReader* received by a local *DataWriter*.
	received_nack_count_change	The incremental change in the number of NACKs from a remote *DataReader* received by a local *DataWriter* since the last time the status was read.
	received_nack_bytes	The number of bytes of NACKs from a remote *DataReader* received by a local *DataWriter*.
	received_nack_bytes_change	The incremental change in the number of bytes of NACKs from a remote DataReader received by a local *DataWriter* since the last time the status was read.

Table 6.7 **DDS_DataWriterProtocolStatus**

Type	Field Name	Description
DDS_LongLong	sent_gap_count	The number of GAPs sent from local *DataWriter* to matching remote *DataReaders*.
	sent_gap_count_change	The incremental change in the number of GAPs sent from local *DataWriter* to matching remote *DataReaders* since the last time the status was read.
	sent_gap_bytes	The number of bytes of GAPs sent from local *DataWriter* to matching remote *DataReaders*.
	sent_gap_bytes_change	The incremental change in the number of bytes of GAPs sent from local *DataWriter* to matching remote *DataReaders* since the last time the status was read.
DDS_LongLong	rejected_sample_count	The number of times a sample is rejected for unanticipated reasons in the send path.
	rejected_sample_count_change	The incremental change in the number of times a sample is rejected due to exceptions in the send path since the last time the status was read.
DDS_Long	send_window_size	Current maximum number of outstanding samples allowed in the *DataWriter's* queue.

Table 6.7 **DDS_DataWriterProtocolStatus**

Type	Field Name	Description
DDS_Sequence Number_t	first_available_sample_ sequence_number	Sequence number of the first available sample in the *DataWriter's* reliability queue.
	last_available_sample_ sequence_number	Sequence number of the last available sample in the *DataWriter's* reliability queue.
	first_unacknowledged_sample_ sequence_number	Sequence number of the first unacknowledged sample in the *DataWriter's* reliability queue.
	first_available_sample_virtual_ sequence_number	Virtual sequence number of the first available sample in the *DataWriter's* reliability queue.
	last_available_sample_virtual_ sequence_number	Virtual sequence number of the last available sample in the *DataWriter's* reliability queue.
	first_unacknowledged_sample_virtual_ sequence_number	Virtual sequence number of the first unacknowledged sample in the *DataWriter's* reliability queue.
	first_unacknowledged_sample_ subscription_handle	Instance Handle of the matching remote *DataReader* for which the *DataWriter* has kept the first available sample in the reliability queue.
	first_unelapsed_keep_duration_ sample_sequence_number	Sequence number of the first sample kept in the *DataWriter's* queue whose keep_duration (applied when disable_positive_acks is set) has not yet elapsed.

6.3.5.3 LIVELINESS_LOST Status

A change to this status indicates that the *DataWriter* failed to signal its liveliness within the time specified by the LIVELINESS QosPolicy (Section 6.5.11).

It is different than the RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.5.8) 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 6.8 on page 6-36.

Table 6.8 **DDS_LivelinessLostStatus**

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the *DataWriter* failed to explicitly signal its liveliness within the liveliness period.
DDS_Long	total_count_change	The change in total_count since the last time the Listener was called or the status was read.

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.

6.3.5.4 OFFERED_DEADLINE_MISSED Status

A change to this status indicates that the *DataWriter* failed to write data within the time period set in its DEADLINE QosPolicy (Section 6.5.4).

The structure for this status appears in Table 6.9.

Table 6.9 **DDS_OfferedDeadlineMissedStatus**

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the *DataWriter* failed to write within its offered deadline.
DDS_Long	total_count_change	The change in total_count since the last time the *Listener* was called or the status was read.
DDS_Instance Handle_t	last_instance_handle	Handle to the last data-instance in the *DataWriter* for which an offered deadline was missed.

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.

6.3.5.5 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 6.10.

Table 6.10 **DDS_OfferedIncompatibleQoSStatus**

Type	Field Name	Description
DDS_Long	total_count	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*.
DDS_Long	total_count_change	The change in total_count since the last time the *Listener* was called or the status was read.
DDS_QosPolicyId_t	last_policy_id	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.)
DDS_ QosPolicyCountSeq	policies	A list containing—for each policy—the total number of times that the *DataWriter* discovered a *DataReader* for the same *Topic* with a requested QoS that is incompatible with that offered by the *DataWriter*.

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.

6.3.5.6 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 data type (implied by having the same *Topic*), and compatible QosPolicies. In addition, if user code has directed *RTI Data Distribution Service* to ignore certain *DataReaders*, then those *DataReaders* will never be matched. See Section 14.4.2 for more on setting up a *DomainParticipant* to ignore specific *DataReaders*.

The structure for this status appears in Table 6.11.

Table 6.11 **DDS_PublicationMatchedStatus**

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the *DataWriter* discovered a "match" with a *DataReader*.
	total_count_change	The change in total_count since the last time the Listener was called or the status was read.
	current_count	The number of *DataReaders* currently matched to the *DataWriter*.
	current_count_peak	The highest value that current_count has reached until now.
	current_count_change	The change in current_count since the last time the listener was called or the status was read.
DDS_Instance Handle_t	last_subscription_handle	Handle to the last *DataReader* that matched the *DataWriter* causing the status to change.

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.

6.3.5.7 RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension)

A change to this status indicates that the number of unacknowledged samples in a reliable *DataWriter*'s cache has reached one of these trigger points:

- ❏ The cache is empty (contains no unacknowledged samples)
- ❏ The cache is full (the number of unacknowledged samples has reached the value specified in **DDS_ResourceLimitsQosPolicy::max_samples**)
- ❏ The number of unacknowledged samples has reached a high or low watermark. See the **high_watermark** and **low_watermark** fields in Table 6.29 of the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

For more about the reliable protocol used by *RTI Data Distribution Service* and specifically, what it means for a sample to be 'unacknowledged,' see Chapter 10: Reliable Communications.

The structure for this status appears in Table 6.12. The supporting structure, **DDS_ReliableWriterCacheEventCount**, is described in Table 6.13.

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.

Table 6.12 **DDS_ReliableWriterCacheChangedStatus**

Type	Field Name	Description
DDS_ReliableWriter CacheEventCount	empty_reliable_writer _cache	How many times the reliable *DataWriter's* cache of unacknowledged samples has become empty.
	full_reliable_writer _cache	How many times the reliable *DataWriter's* cache of unacknowledged samples has become full.
	low_watermark _reliable_writer_cache	How many times the reliable *DataWriter's* cache of unacknowledged samples has fallen to the low watermark.
	high_watermark _reliable_writer_cache	How many times the reliable *DataWriter's* cache of unacknowledged samples has risen to the high watermark.
DDS_Long	unacknowledged_ sample_count	The current number of unacknowledged samples in the *DataWriter's* cache.
	unacknowledged_ sample_count_peak	The highest value that unacknowledged_sample_count has reached until now.

Table 6.13 **DDS_ReliableWriterCacheEventCount**

Type	Field Name	Description
DDS_Long	total_count	The total number of times the event has occurred.
DDS_Long	total_count_change	The number of times the event has occurred since the *Listener* was last invoked or the status read.

6.3.5.8 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 LIVELINESS_CHANGED Status (Section 7.3.6.4) on the *DataReader*. It is different than LIVELINESS_LOST Status (Section 6.3.5.3) status on the *DataWriter,* in that the latter informs the *DataWriter* about its *own* liveliness; this status informs the *DataWriter* about the liveliness of its matched *DataReaders*.

A reliable *DataReader* is considered active by a reliable *DataWriter* with which it is matched if that *DataReader* acknowledges the samples that it has been sent in a timely fashion. For the definition of "timely" in this context, see DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

This status is only used for *DataWriters* whose RELIABILITY QosPolicy (Section 6.5.17) 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 6.14.

Table 6.14 **DDS_ReliableReaderActivityChangedStatus**

Type	Field Name	Description
DDS_Long	active_count	The current number of reliable readers currently matched with this reliable *DataWriter*.
	not_active_count	The number of reliable readers that have been dropped by this reliable *DataWriter* because they failed to send acknowledgements in a timely fashion.
	active_count_change	The change in the number of active reliable *DataReaders* since the *Listener* was last invoked or the status read.
	inactive_count_change	The change in the number of inactive reliable *DataReaders* since the *Listener* was last invoked or the status read.
DDS_Instance Handle_t	last_instance_handle	The instance handle of the last reliable *DataReader* to be determined to be inactive.

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.

6.3.6 Using a Type-Specific DataWriter (FooDataWriter)

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 Chapter 3: Data Types and Data Samples. 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 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 6.3 indicates which operations must be called using *FooDataWriter*. For operations that are not type-specific, you can can call the operation using either a *FooDataWriter* or a *DDSDataWriter* object[1].

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 *DataWriter*s 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()**. For example:

```
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.

6.3.7 Writing Data

The **write()** operation informs *RTI Data Distribution Service* 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*

1. In the C API, the non type-specific operations must be called using a *DDS_DataWriter* pointer.

or even to prevent the data from being sent. Those sending "modes" are configured using the PRESENTATION QosPolicy (Section 6.4.6). 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 Chapter 17: RTI Data Distribution Service Threading Model.

When you call **write()**, *RTI Data Distribution Service* automatically attaches a stamp of the current time that is sent with the 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 The SampleInfo Structure (Section 7.4.5)).

```
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 *RTI Data Distribution Service*.

```
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.

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 Section 6.3.14 and the LIVELINESS QosPolicy (Section 6.5.11). Maintaining liveliness is important for *DataReaders* to know that the *DataWriter* still exists and for the proper behavior of the OWNERSHIP QosPolicy (Section 6.5.13).

See also: Clock Selection (Section 8.6).

6.3.7.1 Blocking During a write()

The **write()** operation may block if the RELIABILITY QosPolicy (Section 6.5.17) *kind* is set to Reliable and the modification would cause data to be lost or cause one of the limits specified in the RESOURCE_LIMITS QosPolicy (Section 6.5.18) to be exceeded. Specifically, **write()** may block in the following situations (note that the list may not be exhaustive), even if its HISTORY QosPolicy (Section 6.5.8) is KEEP_LAST:

- ❏ If **max_samples** < (**max_instances** * **depth**), then in the situation where the max_samples resource limit is exhausted, *RTI Data Distribution Service* may discard samples of some other instance, as long as at least one sample remains for such an instance. If it is still not possible to make space available to store the modification, the writer is allowed to block.
- ❏ If **max_samples** < **max_instances**, then the *DataWriter* may block regardless of the **depth** field in the HISTORY QosPolicy (Section 6.5.8).

This operation may also block when using BEST_EFFORT Reliability (Section 6.5.18) and ASYNCHRONOUS Publish Mode (Section 6.5.16) QoS settings. In this case, the *DataWriter* will queue samples until they are sent by the asynchronous publishing thread. The number of samples that can be stored is determined by the HISTORY QosPolicy (Section 6.5.8). If the asynchronous thread does not send samples fast enough (such as when using a slow FlowController (Section 6.6)), 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 RELIABILITY **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**.

6.3.8 Flushing Batches of Data Samples

The **flush()** operation makes a batch of data samples available to be sent on the network.

```
DDS_ReturnCode_t flush ()
```

If the *DataWriter's* PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.16) **kind** is *not* ASYNCHRONOUS, the batch will be sent on the network immediately in the context of the calling thread.

If the *DataWriter's* PublishModeQosPolicy **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 Blocking During a write() (Section 6.3.7.1).

If this operation does block, the **max_blocking_time** in the RELIABILITY QosPolicy (Section 6.5.17) 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 BATCH QosPolicy (DDS Extension) (Section 6.5.1).

6.3.9 Writing Coherent Sets of Data Samples

A publishing application can request that a set of 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:

1. Call the *Publisher's* **begin_coherent_changes()** operation to indicate the start a coherent set.
2. For each sample in the coherent set: call the *FooDataWriter's* **write()** operation.
3. Call the *Publisher's* **end_coherent_changes()** operation to terminate the set.

Calls to **begin_coherent_changes()** and **end_coherent_changes()** can be nested.

See also: the **coherent_access** field in the PRESENTATION QosPolicy (Section 6.4.6).

6.3.10 Waiting for Acknowledgments

The *DataWriter's* **wait_for_acknowledgments()** operation blocks the calling thread until either all data written by the reliable *DataWriter* is acknowledged by all matched reliable *DataReaders*, or else the duration specified by the **max_wait** parameter elapses, whichever happens first.

```
DDS_ReturnCode_t wait_for_acknowledgments
                                    (const DDS_Duration_t &  max_wait)
```

This operation returns **DDS_RETCODE_OK** if all the samples were acknowledged, or **DDS_RETCODE_TIMEOUT** if the **max_wait** duration expired first.

If the *DataWriter* does not have its RELIABILITY QosPolicy (Section 6.5.17) **kind** set to RELIABLE, the operation will immediately return **DDS_RETCODE_OK**.

There is a similar operation available at the *Publisher* level, see Section 6.2.7.

The reliability protocol used by *RTI Data Distribution Service* is discussed in Chapter 10: Reliable Communications.

6.3.11 Managing Data Instances (Working with Keyed Data Types)

This section applies only to data types that use keys, see Samples, Instances, and Keys (Section 2.2.2). Using the following operations for non-keyed types has no effect.

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 6.10.

Figure 6.10 **Data Type with a Key**

```
typedef struct Flight {
    long   flightId; //@key
    string departureAirport;
    string arrivalAirport;
    Time_t departureTime;
    Time_t estimatedArrivalTime;
    Location_t currentPosition;
};
```

If the data type has some fields that act as a 'key,' the *Topic* essentially defines a collection of data-instances whose values can be independently maintained. In Figure 6.10, the **flightId** is the 'key'. Different flights will have different values for the key. Each flight is an instance of the *Topic*. Each **write()** will update the information about a single flight. *DataReaders* can be informed when new flights appear or old ones disappear.

Since the key fields are contained within the data structure, *RTI Data Distribution Service* could examine the key fields each time it needs to determine which data-instance is being modified. However, for performance and semantic reasons, it is better for your application to declare all the data-instances it intends to modify—prior to actually writing any samples. This is known as *registration*, described below in Section 6.3.11.1.

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.

6.3.11.1 Registering and Unregistering Instances

If your data type has a key, you may improve performance by registering an instance (data associated with a particular value of the key) before you write data for the instance. You can do this for any number of instances up the maximum number of instances configured in the *DataWriter*'s RESOURCE_LIMITS QosPolicy (Section 6.5.18). Instance registration is completely optional.

Registration tells *RTI Data Distribution Service* that you are about to modify (write or dispose of) a specific instance. This allows *RTI Data Distribution Service* to pre-configure itself to process that particular instance, which can improve performance.

If you write without registering, you can pass the NIL instance handle as part of the **write()** call.

If you register the instance first, *RTI Data Distribution Service* can look up the instance beforehand and return a handle to that instance. Then when you pass this handle to the **write()** operation, *RTI Data Distribution Service* no longer needs to analyze the data to check what instance it is for. Instead, it can directly update the instance pointed to by the instance handle.

In summary, by registering an instance, all subsequent **write()** calls to that instance become more efficient. If you only plan to write once to a particular instance, registration does not 'buy' you much in performance, but in general, it is good practice.

To register an instance, use the *DataWriter*'s **register_instance()** operation. For best performance, it should be invoked prior to calling any operation that modifies the instance, such as **write()**, **write_w_timestamp()**, **dispose()**, or **dispose_w_timestamp()**.

When you are done using that instance, you can unregister it. To unregister an instance, use the *DataWriter*'s **unregister_instance()** operation. Unregistering tells *RTI Data Distribution Service* that the *DataWriter* does not intend to modify that data-instance anymore, allowing *RTI Data Distribution Service* to recover any resources it allocated for the instance. It does not delete the instance; that is done with the **dispose_instance()** operation, see Section 6.3.11.2. **unregister_instance()** should only be used on instances that have been previously registered. The use of these operations is illustrated in Figure 6.11.

Once an instance has been unregistered, and assuming that no other *DataWriters* are writing values for the instance, the matched *DataReader*'s will eventually get an indication that the instance no longer has any *DataWriter*s. This is communicated to the *DataReaders* by means of the **DDS_SampleInfo** that accompanies each data-sample (see Section 7.4.5). Once there are no *DataWriter*s for the instance, the *DataReader* will see the value of **DDS_InstanceStateKind** for that instance to be **NOT_ALIVE_NO_WRITERS**.

The **unregister_instance()** operation may affect the ownership of the data instance (see the OWNERSHIP QosPolicy (Section 6.5.13)). 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.

The **unregister_instance()** operation indicates only that a particular *DataWriter* no longer has anything to say about the instance.

Figure 6.11 **Registering an Instance**

```
Flight myFlight;

// writer is a previously-created FlightDataWriter
myFlight.flightId = 265;
DDS_InstanceHandle_t fl265Handle =
                         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.estimatedArrivalTime = {130200, 0};
myFlight.currentPosition      = { {37, 20}, {121, 53} };

if (writer->write(myFlight, fl265Handle) != DDS_RETCODE_OK) {
    // ... handle error
}
...

// Once we are done updating the flight, it can be unregistered
if (writer->unregister_instance(myFlight, fl265Handle) !=
                                               DDS_RETCODE_OK) {
    // ... handle error
}
```

Note that this is different than the **dispose()** operation discussed in the next section, which informs *DataReaders* that the data-instance is no longer "alive." The state of an instance is stored in the **DDS_SampleInfo** structure that accompanies each sample of data that is received by a *DataReader*. User code can access the instance state to see if an instance is "alive"—meaning there is at least one *DataWriter* that is publishing samples for the instance, see Instance States (Section 7.4.5.4).

6.3.11.2 Disposing of Data

The **dispose()** operation informs *DataReaders* that, as far as the *DataWriter* knows, the data-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.

For example, in a flight tracking system, when a flight lands, a *DataWriter* may dispose the data-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 that this is different than **unregister_instance()** (Section 6.3.11.1), which indicates only that a particular *DataWriter* no longer wishes to modify an instance—an important distinction if there are multiple writers on the same instance.

If a particular instance is never disposed, its instance state will eventually change from **ALIVE** to **NOT_ALIVE_NO_WRITERS** once all the *DataWriter*s that were writing that instance unregister the instance or lose their liveliness. For more information on *DataWriter* liveliness, see the LIVELINESS QosPolicy (Section 6.5.11).

See also: Propagating Serialized Keys with Disposed-Instance Notifications (Section 6.5.2.5).

6.3.11.3 Looking Up an Instance Handle

Some operations, such as **write()**, require an **instance_handle** parameter. If you need to get such as 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 for keyed data types.

```
DDS_InstanceHandle_t lookup_instance (const Foo & key_holder)
```

The instance must have already been registered (see Section 6.3.11.1). If the instance is not registered, this operation returns **DDS_HANDLE_NIL**.

6.3.11.4 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 Samples, Instances, and Keys (Section 2.2.2) for more information.

Following our example in Figure 6.11 on page 6-47, **register_instance()** returns a **DDS_InstanceHandle_t** (**fl265Handle**) 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: Propagating Serialized Keys with Disposed-Instance Notifications (Section 6.5.2.5).

6.3.12 Setting DataWriter QosPolicies

The *DataWriter's* QosPolicies control its resources and behavior.

The **DDS_DataWriterQos** structure has the following format:

```
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;

    // extensions to the DDS standard:
    DDS_DataWriterResourceLimitsQosPolicy writer_resource_limits;
    DDS_DataWriterProtocolQosPolicy      protocol;
    DDS_TransportSelectionQosPolicy      transport_selection;
    DDS_TransportUnicastQosPolicy        unicast;
    DDS_TypeSupportQosPolicy             type_support;
    DDS_PublishModeQosPolicy             publish_mode;
    DDS_PropertyQosPolicy                property;
    DDS_BatchQosPolicy                   batch;
    DDS_MultiChannelQosPolicy            multi_channel;
} DDS_DataWriterQos;
```

Note: set_qos() cannot always be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

Table 6.15 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 online documentation.

Table 6.15 **DataWriter QosPolicies**

QosPolicy	Description
Batch	Specifies and configures the mechanism that allows *RTI Data Distribution Service* to collect multiple 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 Section 6.5.1.
DataWriterProtocol	This QosPolicy configures the DDS on-the-network protocol, RTPS. See Section 6.5.2.
DataWriterResourceLimits	Controls how many threads can concurrently block on a write() call of this *DataWriter*. See Section 6.5.3.
Deadline	For a *DataReader*, specifies the maximum expected elapsed time between arriving data samples. For a *DataWriter*, specifies a commitment to publish samples with no greater elapsed time between them. See Section 6.5.4.
DestinationOrder	Controls how *RTI Data Distribution Service* 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 Section 6.5.5.
Durability	Specifies whether or not *RTI Data Distribution Service* will store and deliver data that were previously published to new *DataReaders*. See Section 6.5.6.
DurabilityService	Various settings to configure the external Persistence Service used by *RTI Data Distribution Service* for *DataWriters* with a Durability QoS setting of Persistent Durability. See Section 6.5.7.
History	Specifies how much data must to stored by *RTI Data Distribution Service*for the *DataWriter* or *DataReader*. This QosPolicy affects the RELIABILITY QosPolicy (Section 6.5.17) as well as the DURABILITY QosPolicy (Section 6.5.6). See Section 6.5.8.
LatencyBudget	Suggestion to DDS on how much time is allowed to deliver data. See Section 6.5.9.
Lifespan	Specifies how long *RTI Data Distribution Service* should consider data sent by an user application to be valid. See Section 6.5.10.
Liveliness	Specifies and configures the mechanism that allows *DataReaders* to detect when *DataWriter*s become disconnected or "dead." See Section 6.5.11.
MultiChannel	Configures a *DataWriter*'s ability to send data on different multicast groups (addresses) based on the value of the data. See Section 6.5.12.
Ownership	Along with OwnershipStrength, specifies if *DataReaders* for a topic can receive data from multiple *DataWriters* at the same time. See Section 6.5.13.
OwnershipStrength	Used to arbitrate among multiple *DataWriters* of the same instance of a Topic when Ownership QosPolicy is EXLUSIVE. See Section 6.5.14.

Table 6.15 **DataWriter QosPolicies**

QosPolicy	Description
Partition	Adds string identifiers that are used for matching *DataReaders* and *DataWriters* for the same *Topic*. See Section 6.4.5.
Property	Stores name/value (string) pairs that can be used to configure certain parameters of *RTI Data Distribution Service* 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 Section 6.5.15.
PublishMode	Specifies how *RTI Data Distribution Service* 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 *RTI Data Distribution Service* to use its own thread to send the data. See Section 6.5.16.
Reliability	Specifies whether or not DDS will deliver data reliably. See Section 6.5.17.
ResourceLimits	Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See Section 6.5.18.
TransportPriority	Set by a *DataWriter* to tell DDS that the data being sent is a different "priority" than other data. See Section 6.5.19.
TransportSelection	Allows you to select which physical transports a *DataWriter* or *DataReader* may use to send or receive its data. See Section 6.5.20.
TransportUnicast	Specifies a subset of transports and port number that can be used by an Entity to receive data. See Section 6.5.21.
TypeSupport	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 Section 6.5.22.
UserData	Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data. See Section 6.5.23.
WriterDataLifeCycle	Controls how a *DataWriter* handles the lifecycle of the instances (keys) that the *DataWriter* is registered to manage. See Section 6.5.24.

Many of the *DataWriter* QosPolicies also apply to *DataReaders* (see Section 7.3). 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 QoS Requested vs. Offered Compatibility—the RxO Property (Section 4.2.1).

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 DDS Entities, described in Section 4.1.7.3.

6.3.12.1 Configuring QoS Settings when the DataWriter is Created

As described in Creating DataWriters (Section 6.3.1), 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 6.9 on page 6-25, we saw 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 Section 4.1.7, this is a general pattern that applies to the construction of all DDS *Entities*.
- To create a *DataWriter* with non-default QoS without using a QoS Profile, see the example code in Figure 6.12 on page 6-53. 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 6.13 on page 6-53.
- 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 6.14 on page 6-54.

For more information, see Creating DataWriters (Section 6.3.1) and Chapter 15: Configuring QoS with XML.

Figure 6.12 **Creating a DataWriter with Modified QosPolicies (not from a profile)**

```
DDS_DataWriterQos writer_qos;[1]

// 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
DDSDataWriter * 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);
```

1. Note: In C, you must initialize the QoS structures before they are used, see Section 4.2.2.

Figure 6.13 **Creating a DataWriter with a QoS Profile**

```
// 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 6.14 **Getting QoS Values from a Profile, Changing QoS Values, Creating a DataWriter with Modified QoS Values**

```
DDS_DataWriterQos writer_qos;[1]

// 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
}
```

1. Note: In C, you must initialize the QoS structures before they are used, see Section 4.2.2.

6.3.12.2 Changing QoS Settings After the DataWriter Has Been Created

There are 2 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 6.15. 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 6.16. For more information, see Chapter 15: Configuring QoS with XML.

Figure 6.15 **Changing the QoS of an Existing DataWriter (without a QoS Profile)**

```
DDS_DataWriterQos writer_qos;[1]

// 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
}
```

1. For the C API, you need to use DDS_ParticipantQos_INITIALIZER or DDS_ParticipantQos_initialize(). See Special QosPolicy Handling Considerations for C (Section 4.2.2)

Figure 6.16 **Changing the QoS of an Existing DataWriter with a QoS Profile**

```
retcode = writer->set_qos_with_profile(
                  "WriterProfileLibrary","WriterProfile");

if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

6.3.12.3 Using a Topic's QoS to Initialize a DataWriter's QoS

Several *DataWriter* QosPolicies can also be found in the QosPolicies for *Topics* (see Section 5.1.3). 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. *RTI Data Distribution Service* 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 6.17.

Figure 6.17 **Copying Selected QoS from a Topic when Creating a DataWriter**

```
DDS_DataWriterQos writer_qos;[1]
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 the topic QoS to the 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);
```

1. Note in C, you must initialize the QoS structures before they are used, see Section 4.2.2.

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 6.18.

Figure 6.18 **Copying all QoS from a Topic when Creating a DataWriter**

```
DDS_DataWriterQos writer_qos;[1]
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 QosPolicies from topic's qos 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);
```

1. Note in C, you must initialize the QoS structures before they are used, see Section 4.2.2.

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 6.19 gives an example of how to do this.

Because this is a common pattern, *RTI Data Distribution Service* 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 6.20 shows how the macro is used.

The code fragments shown in Figure 6.19 and Figure 6.20 result in identical QoS settings for the created *DataWriter*.

Figure 6.19 **Combining Default Topic and DataWriter QoS (Option 1)**

```
DDS_DataWriterQos writer_qos;[1]
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
}
if (publisher->copy_from_topic_qos(writer_qos, topic_qos) !=
                                                 DDS_RETCODE_OK) {
    // handle error
}
// Create the DataWriter with the combined QoS
DDSDataWriter* writer =
           publisher->create_datawriter(topic, writer_qos,
                                 writer_listener,
                                 DDS_STATUS_MASK_ALL);
```

1. Note in C, you must initialize the QoS structures before they are used, see Section 4.2.2.

Figure 6.20 **Combining Default Topic and DataWriter QoS (Option 2)**

```
// topic, publisher, writer_listener already created

DDSDataWriter* writer = publisher->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 Section 4.1.7.

6.3.13 Navigating Relationships Among Entities

6.3.13.1 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 *RTI Data Distribution Service* will send the data written by the *DataWriter*).

- **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.

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 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 *RTI Data Distribution Service* uses to communicate with matching *DataReaders*. (See Locator Format (Section 12.2.1.1).)

You can also get the DATA_WRITER_PROTOCOL_STATUS for matching subscriptions with these operations (see Section 6.3.5.2):

- **get_matched_subscription_datawriter_protocol_status()**
- **get_matched_subscription_datawriter_protocol_status_by_locator()**

Notes:

- Status/data for a matched subscription is only kept while the matched subscription is alive. Once a matched subscription is no longer alive, its status is deleted. If you try to get the status/data for a matched subscription that is no longer alive, the 'get status' or ' get data' call will return an error.
- *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 Section 14.4.2 for more on **ignore_subscription()**.

- ❑ The **get_matched_subscription_data()** operation does not retrieve the following information from builtin-topic data structures: **type_code**, **property**, and **content_filter_property**. This information is available through the **on_data_available()** callback (if a DataReaderListener is installed on the SubscriptionBuiltinTopicDataDataReader).

6.3.13.2 Finding Related 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.

6.3.14 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 LIVELINESS QosPolicy (Section 6.5.11) is **MANUAL_BY_PARTICIPANT** or **MANUAL_BY_TOPIC**.

How *DataReaders* determine if *DataWriters* are alive is configured using the LIVELINESS QosPolicy (Section 6.5.11). 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 *RTI Data Distribution Service* 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.

6.4 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 6.2 on page 6-10 provides a quick reference. They are presented here in alphabetical order.

- ❏ ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1)
- ❏ ENTITYFACTORY QosPolicy (Section 6.4.2)
- ❏ EXCLUSIVE_AREA QosPolicy (DDS Extension) (Section 6.4.3)
- ❏ GROUP_DATA QosPolicy (Section 6.4.4)
- ❏ PARTITION QosPolicy (Section 6.4.5)
- ❏ PRESENTATION QosPolicy (Section 6.4.6)

6.4.1 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension)

This QosPolicy is used to enable or disable asynchronous publishing and asynchronous batch flushing for the *Publisher.*

This QosPolicy can be used to reduce amount of time spent in the user thread to send data. You can use it to send *large* data reliably. *Large* in this context means that the data that cannot be sent as a single packet by a transport. For example, to send data larger than 63K reliably using UDP/IP, one must configure DDS to send the data using asynchronous *Publishers*.

If so configured, the *Publisher* will spawn two threads, one for asynchronous publishing and one for asynchronous batch flushing. The asynchronous publisher thread will be shared by all *DataWriters* (belonging to this Publisher) that have their PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.16) **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 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 BATCH QosPolicy (DDS Extension) (Section 6.5.1). This thread will only be spawned when the first of these *DataWriters* is enabled.

This QosPolicy allows you to adjust the asynchronous publishing and asynchronous batch flushing 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* Flow Controllers (DDS Extension) (Section 6.6). From the point of view of the FlowController, a batch is treated like one large sample.

RTI Data Distribution Service will sometimes coalesce multiple samples into a single network datagram. For example, samples buffered by a FlowController or sent in response to a negative acknowledgement (NACK) may be coalesced. This behavior is distinct from sample batching. 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 samples written by the *same DataWriter* and intended for the *same destination* will be coalesced.

This QoS includes the members in Table 6.16.

Table 6.16 **DDS_AsynchronousPublisherQosPolicy**

Type	Field Name	Description
DDS_Boolean	disable_asynchronous_write	Disables asynchronous publishing. To write asynchronously, this field must be FALSE (the default).
DDS_ThreadSettings_t	thread	Settings for the publishing thread. These settings are OS-dependent.
DDS_Boolean	disable_asynchronous_batch	Disables asynchronous batch flushing. To flush asynchronously, this field must be FALSE (the default).
DDS_ThreadSettings_t	asynchronous_batch_thread	Settings for the asynchronous batch flushing thread. These settings are OS-dependent.

6.4.1.1 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.

6.4.1.2 Related QosPolicies

❏ If **disable_asynchronous_write** is TRUE (not the default), then any *DataWriters* created from this *Publisher* must have their PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.16) **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 BATCH QosPolicy (DDS Extension) (Section 6.5.1) set to DURATION_INFINITE. (Otherwise **create_datawriter()** will return INCONSISTENT_QOS.)

- ❏ *DataWriters* configured to use the MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12) do not support asynchronous publishing; an error is returned if a multi-channel *DataWriter* is configured for asynchronous publishing.

6.4.1.3 Applicable Entities

- ❏ Publishers (Section 6.2)

6.4.1.4 System Resource Considerations

Two threads can potentially be created.

For asynchronous publishing, system resource usage depends on the activity of the asynchronous thread controlled by the FlowController (see Flow Controllers (DDS Extension) (Section 6.6)).

For asynchronous batch flushing, system resource usage depends on the activity of the asynchronous thread controlled by **max_flush_delay** in BATCH QosPolicy (DDS Extension) (Section 6.5.1).

6.4.2 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 *DataWriter*s, 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, *RTI Data Distribution Service* will not send data if the **write()** operation is called on a disabled *DataWriter*, nor will *RTI Data Distribution Service* deliver data to a disabled *DataReader*. You can only enable a disabled entity. Once an entity is enabled, you cannot disable it, see Section 4.1.2 about the **enable()** method.

The ENTITYFACTORY contains only one member, as illustrated in Table 6.17.

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*

Table 6.17 **DDS_EntityFactoryQosPolicy**

Type	Field Name	Description
DDS_Boolean	autoenable_created_entities	DDS_BOOLEAN_TRUE: enable entities when they are created DDS_BOOLEAN_FALSE: do not enable entities when they are created

is configured to auto-enable created entities, then all *DataWriter*s 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 Section 6.4.2.1 for an example of how to set this policy.

There are various reasons why you may want to create entities in the disabled state:

❑ 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* applications exist. However, you may want to configure the builtin 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 builtin 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*, *RTI Data Distribution Service* will immediately search for matching *DataWriters* and callback 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 Section 4.1.2.

❑ 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, *RTI Data Distribution Service* can be more efficient in bundling multiple declarations into a single packet when you enable all entities at the same time.

See Section 4.1.2 for more information about enabled/disabled entities.

6.4.2.1 Example

The code in Figure 6.21 illustrates how to use the ENTITYFACTORY QoS.

Figure 6.21 **Configuring a Publisher so that New DataWriters are Disabled**

```
DDS_PublisherQos publisher_qos;[1]

// 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();
```

1. Note in C, you must initialize the QoS structures before they are used, see Section 4.2.2.

6.4.2.2 Properties

This QosPolicy can be modified at any time.

It can be set differently on the publishing and subscribing sides.

6.4.2.3 Related QosPolicies

This QosPolicy does not interact with any other policies.

6.4.2.4 Applicable Entities

- ❑ DomainParticipantFactory (Section 8.2)
- ❑ DomainParticipants (Section 8.3)
- ❑ Publishers (Section 6.2)
- ❑ Subscribers (Section 7.2)

6.4.2.5 System Resource Considerations

This QosPolicy does not significantly impact the use of system resources.

6.4.3 EXCLUSIVE_AREA QosPolicy (DDS Extension)

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 *RTI Data Distribution Service* code.

EAs allow *RTI Data Distribution Service* 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 Exclusive Areas (EAs) (Section 4.5) 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 *RTI Data Distribution Service* functions for any Entities from their own threads at any time.

The EXCLUSIVE_AREA includes a single member, as listed in Table 6.18. For the default value, please refer to the online documentation.

Table 6.18 **DDS_ExclusiveAreaQosPolicy**

Type	Field Name	Description
DDS_Boolean	use_shared_exclusive_area	DDS_BOOLEAN_FALSE: subordinates will not use the same EA DDS_BOOLEAN_TRUE: subordinates will use the same EA

The implications and restrictions of using a private or shared EA are discussed in Section 4.5. The basic trade-off is concurrency versus restrictions on which methods can be called in user, listener, callback functions. To summarize:

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 *RTI Data Distribution Service* operations (see Operations Allowed within Listener Callbacks (Section 4.4.5)) 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 Exclusive Areas (EAs) (Section 4.5) for full details).

6.4.3.1 Example

The code in Figure 6.22 illustrates how to change the EXCLUSIVE_AREA policy.

Figure 6.22 **Creating a Publisher with a Shared Exclusive Area**

```
DDS_PublisherQos publisher_qos;[1]

// 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);
```

1. Note in C, you must initialize the QoS structures before they are used, see Section 4.2.2.

6.4.3.2 Properties

This QosPolicy cannot be modified after the Entity has been created.

It can be set differently on the publishing and subscribing sides.

6.4.3.3 Related QosPolicies

This QosPolicy does not interact with any other policies.

6.4.3.4 Applicable Entities

- ❑ Publishers (Section 6.2)
- ❑ Subscribers (Section 7.2)

6.4.3.5 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.

6.4.4 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 Chapter 12: Discovery) using builtin-topics (see Chapter 14: Built-In Topics). How this information is used will be up to user code. *RTI Data Distribution Service* 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_qos()** 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 *RTI Data Distribution Service* 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 Chapter 14: Built-In Topics).

The structure for the GROUP_DATA QosPolicy includes just one field, as seen in Table 6.19. 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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4).

Table 6.19 **DDS_GroupDataQosPolicy**

Type	Field Name	Description
DDS_OctetSeq	value	Empty by default

This policy is similar to the USER_DATA QosPolicy (Section 6.5.23) and TOPIC_DATA QosPolicy (Section 5.2.1) that apply to other types of Entities.

6.4.4.1 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 *RTI Data Distribution Service* 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 Chapter 14: Built-In Topics). 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 *RTI Data Distribution Service* to communicate. See Figure 14.2, "Ignoring Publications," on page 14-17 for an example of how to do this.

The code in Figure 6.23 illustrates how to change the GROUP_DATA policy.

6.4.4.2 Properties

This QosPolicy can be modified at any time.

It can be set differently on the publishing and subscribing sides.

Figure 6.23 **Creating a Publisher with GROUP_DATA**

```
DDS_PublisherQos publisher_qos;1
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 that domainparticipant and publisher_listener
// are 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);
```

1. Note in C, you must initialize the QoS structures before they are used, see Section 4.2.2.

6.4.4.3 Related QosPolicies

- TOPIC_DATA QosPolicy (Section 5.2.1)
- USER_DATA QosPolicy (Section 6.5.23)
- DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)

6.4.4.4 Applicable Entities

- ❏ Publishers (Section 6.2)
- ❏ Subscribers (Section 7.2)

6.4.4.5 System Resource Considerations

As mentioned earlier, 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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4). Because *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 (Section 5.2.1) and USER_DATA (Section 6.5.23) QosPolicies.

6.4.5 PARTITION QosPolicy

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 QosPolicies 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.

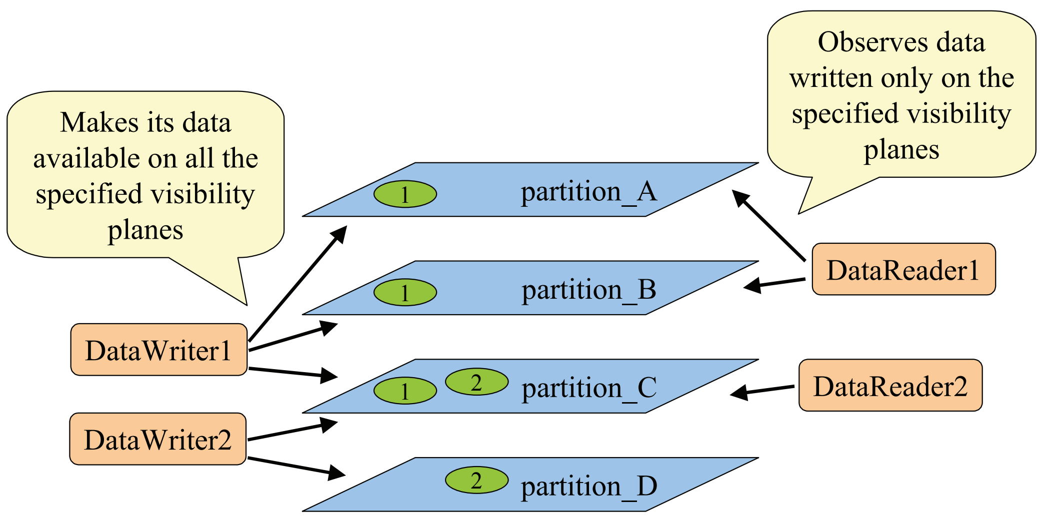
The PARTITION QoS applies to *Publishers* and *Subscribers*, therefore the *DataWriters* and *DataReaders* belong to the partitions as set on the *Publishers* and *Subscribers* that created them. The mechanism implementing the PARTITION QoS is relatively lightweight, and membership in a partition can 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 more than one partition name in common in the PARTITION QoS set on their *Publishers* or *Subscribers*.

Conceptually each partition name can be thought of as defining a "visibility plane" within the DDS domain. *DataWriter*s 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 6.24 illustrates the concept of PARTITION QoS. In this figure, all *DataWriter*s and *DataReaders* belong to the same 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 6.24 **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 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 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, *RTI Data Distribution Service* 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 data samples written by *DataWriters* 1 and 2.

6.4.5.1 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 domain. That is, their respective *DomainParticipants* are bound to the same domain ID (see Section 8.3.1).
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 Section 14.4).
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.

6.4.5.2 Pattern Matching for PARTITION Names

You may also add strings that are regular expressions[1] 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.

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 6.20. 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.

Table 6.20 **DDS_PartitionQosPolicy**

Type	Field Name	Description
DDS_StringSeq	name	Empty by default. There can be up to 64 names, with a maximum of 256 characters summed across all names.

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.

1. As defined by the POSIX fnmatch API (1003.2-1992 section B.6).

6.4.5.3 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.

The code in Figure 6.25 illustrates how to change the PARTITION policy.

Figure 6.25 **Setting Partition Names on a Publisher**

```
DDS_PublisherQos publisher_qos;[1]
// domain, publisher_listener have been previously created

if (participant->get_default_publisher_qos(publisher_qos) !=
                                                DDS_RETCODE_OK) {
    // handle error
}

// Set the partition 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");

DDSPublisher* publisher =
            participant->create_publisher(publisher_qos,
                            publisher_listener, DDS_STATUS_MASK_ALL);
```

1. Note in C, you must initialize the QoS structures before they are used, see Section 4.2.2.

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 6.21.

Table 6.21 **Example of Using Location-Based Partitions**

Publisher Partitions	Subscriber Partitions	Result
Specify a single partition name using the pattern: "<country>/<state>/<city>"	Specify multiple partition names, one per region of interest	Limits the visibility of the data to Subscribers that express interest in the geographical region.
"USA/California/Santa Clara"	(Subscriber participant is irrelevant here.)	Send only information for Santa Clara, California.
(Publisher partition is irrelevant here.)	"USA/California/Santa Clara"	Receive only information for Santa Clara, California.
	"USA/California/Santa Clara" "USA/California/Sunnyvale"	Receive information for Santa Clara or Sunnyvale, California.
	"USA/California/*" "USA/Nevada/*"	Receive information for California or Nevada.
	"USA/California/*" "USA/Nevada/Reno" "USA/Nevada/Las Vegas"	Receive information for California and two cities in Nevada.

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 6.22.

A slight variation of this pattern could be used to confine the information based on security levels.

Table 6.22 **Example of Access-Control Group Partitions**

Publisher Partitions	Subscriber Partitions	Result
Specify several partition names, one per group that is allowed access:	Specify multiple partition names, one per group to which the Subscriber belongs.	Limits the visibility of the data to Subscribers that belong to the access-groups specified by the Publisher.
"payroll" "financial"	(Subscriber participant is irrelevant here.)	Makes information available only to Subscribers that have access to either financial or payroll information.
(Publisher participant is irrelevant here.)	"executives" "financial"	Gain access to information that is intended for executives or people with access to the finances.

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.

6.4.5.4 Properties

This QosPolicy can be modified at any time.

Strictly speaking, this QosPolicy does not have request-offered semantics, although it is matched between *DataWriter*s and *DataReaders*, and communication is established only if there is a match between partition names.

6.4.5.5 Related QosPolicies

- DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4).

6.4.5.6 Applicable Entities

- Publishers (Section 6.2)
- Subscribers (Section 7.2)

6.4.5.7 System Resource Considerations

Partition names are propagated along with the declarations of the *DataReaders* and the *DataWriter*s and can be examined by user code through built-in topics (see Chapter 14: Built-In Topics). 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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4). 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 *RTI Data Distribution Service* 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 (Section 6.4.4), USER_DATA (Section 6.5.23), and TOPIC_DATA (Section 5.2.1) QosPolicies.

6.4.6 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 topic to be presented to the receiving *DataReader* only after ALL of 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 *RTI Data Distribution Service* to present the data in keyed or instance order.

The Presentation QosPolicy allows you to specify different scopes of presentation, within a topic, across instances of a topic, and even across different topics of a publisher (although this last option is not currently supported by RTI). It also controls whether or not a set of changes within the scope is delivered at the same time or can be delivered as soon as each element is received.

There are three components to this QoS, the boolean flag **coherent_access,** the boolean flag **ordered_access**, and an enumerated setting for the **access_scope**. The structure used is shown in Table 6.23.

Table 6.23 **DDS_PresentationQosPolicy**

Type	Field Name	Description
DDS_Presentation_ QosPolicyAccessScopeKind	access_scope	Controls the granularity used when coherent_access and/or ordered_access are TRUE. If both coherent_access and ordered_access are FALSE, **access_scope's** setting has no effect. DDS_INSTANCE_PRESENTATION_QOS: queue is ordered/sorted per instance DDS_TOPIC_PRESENTATION_QOS: queue is ordered/sorted per topic (across all instances)
DDS_Boolean	coherent_access	Controls whether *RTI Data Distribution Service* will preserve the groupings of changes made by the publishing application by means of **begin_coherent_changes()** and **end_coherent_changes()**. DDS_BOOLEAN_FALSE: Coherency is not preserved. The value of access_scope is ignored. DDS_BOOLEAN_TRUE: Changes made to instances within each *DataWriter* will be available to the *DataReader* as a coherent set, based on the value of **access_scope**.
DDS_Boolean	ordered_access	Controls whether *RTI Data Distribution Service* will preserve the order of changes. DDS_BOOLEAN_FALSE: The order of samples is only preserved for each instance, not across instances. The value of **access_scope** is ignored. DDS_BOOLEAN_TRUE: The order of samples from a *DataWriter* is preserved, based on the value set in **access_scope**.

6.4.6.1 Coherent Access

A 'coherent set' is a set of 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 samples that belong to a coherent set.

To send a coherent set of data samples, the publishing application uses the *Publisher's* **begin_coherent_changes()** and **end_coherent_changes()** operations (see Writing Coherent Sets of Data Samples (Section 6.3.9)).

If **coherent_access** is TRUE, then the **access_scope** controls the maximum extent of the coherent changes, as follows:

- ❏ If **access_scope** is set to INSTANCE, the use of **begin_coherent_changes()** and **end_coherent_changes()** has no effect on how the subscriber can access the data. This is because, with the scope limited to each instance, changes to separate instances are considered independent and thus cannot be grouped by a coherent change.
- ❏ If **access_scope** is set to TOPIC, then coherent changes (indicated by their enclosure within calls to **begin_coherent_changes()** and **end_coherent_changes()**) will be made available as such to each remote *DataReader* independently. That is, changes made to instances within the each individual *DataWriter* will be available as a coherent set with respect to other changes to instances in that same *DataWriter*, but will not be grouped with changes made to instances belonging to a different *DataWriter*.

6.4.6.2 Ordered Access

For a non-keyed *Topic*, data samples from a single *DataWriter* will be stored in the *DataReader's* queue in the order they were sent (as one would expect). However, recall that a *Topic* may have multiple data instances only if it uses a keyed data type, see Samples, Instances, and Keys (Section 2.2.2). So for keyed *Topics*, a single *DataWriter* may create instances, modify the value of instances, or delete instances. Each time, a data sample (including pseudo or meta-samples for the creation and deletion events) will be sent and must be stored in the *DataReader*'s queue.

By default, instances are treated as “lightweight” *Topics* in that the data samples that pertain to a particular instance are stored in the order that they are sent by the *DataWriter*. This includes the samples that represent the events of instance creation and deletion.

However, because data samples across different instances of the same *Topic* are stored in the same queue for a *DataReader*, you may want to control whether or not the data samples are stored in the order sent by the *DataWriter*—independent of the instance for which the data sample applies. By default, the *Subscriber* does not try to sort the data samples across instances. Since data packets may be delivered out of order by the network, you cannot correlate the data samples between instances, even though they were generated by the same *DataWriter*.

For example, if a *DataWriter* registers three instances, this will generate three packets that indicate the creation of those instances, resulting in three meta-data samples stored in a *DataReader*’s queue. By default, the *Subscriber* will not try to order these samples if they were delivered out of order. You may see the sample that indicates the creation of the third instance before you see the sample that indicates the creation of the first.

However, you can set this QoS so that the *DataReader* will see all events—creation, modification, deletion—across all instances written by a *DataWriter* in the order that they occurred. Note that this applies to data coming from a single *DataWriter* only. The sorting of data samples between *DataWriters* are not affected by this QoS. See DESTINATION_ORDER QosPolicy (Section 6.5.5) if you would like to control how *RTI Data Distribution Service* sorts data across multiple *DataWriters* of the same *Topic*.

If **ordered_access** is TRUE, then the **access_scope** controls the scope of the ordering, as follows:

- If **access_scope** is set to INSTANCE, the relative order of samples sent by a *DataWriter* is only preserved on an per-instance basis. If two 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 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 set to TOPIC**,** the relative order of samples sent by a *DataWriter* is preserved for all samples of all instances. The coherent grouping and/or order in which samples appear in the *DataReader*’s queue is consistent with the grouping/order in which the changes occurred—even if the samples affect different instances.

The data stored in the *DataReader* is accessed by the *DataReader*’s **read()/take()** APIs. The application does not have to access the data samples in the same order as they are stored in the queue. How the application actually gets the data from the *DataReader* is ultimately under the control of the user code, see Using DataReaders to Access Data (Read & Take) (Section 7.4).

6.4.6.3 Example

Coherency is useful in cases where the values are inter-related (for example, if there are two data-instances representing the altitude and velocity vector of the same aircraft and both are changed, it may be useful to communicate those values in a way the reader can see both together; otherwise, it may e.g., erroneously interpret that the aircraft is on a collision course).

Ordered access is useful when you need to ensure that samples appear on the *DataReader's* in the order sent by the *DataWriter*.

To illustrate the effect of the PRESENTATION QosPolicy, assume the following sequence of 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 samples have been propagated to the *DataReader* history queue before your application invokes the **read()** operation. The data-sample sequence returned will depend on the setting of the PRESENTATION QoS as shown in Table 6.24.

Table 6.24 **Effect of ordered_access in the PRESENTATION QoS**

PRESENTATION QoS	Sequence retrieved via “read()”. Order sent was {A1, B1, C1, A2, B2, C2} Order received was {A1, A2, B1, B2, C1, C2}
ordered_access = FALSE access_scope = <any>	{A1, A2, B1, B2, C1, C2}
ordered_access = TRUE access_scope = INSTANCE	{A1, A2, B1, B2, C1, C2}
ordered_access = TRUE access_scope = TOPIC	{A1, B1, C1, A2, B2, C2}

6.4.6.4 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 6.25 for **ordered_access** and Table 6.25 for **coherent_access**.

Table 6.25 **Valid Combinations of Presentation Ordered Access and Access Scope**

{ordered_access/ access_scope}		Subscriber requests:			
		False/Instance	False/Topic	True/Instance	True/Topic
Publisher offers:	False/Instance	✔	incompatible	incompatible	incompatible
	False/Topic	✔	✔	incompatible	incompatible
	True/Instance	✔	incompatible	✔	incompatible
	True/Topic	✔	✔	✔	✔

Table 6.26 **Valid Combinations of Presentation Coherent Access and Access Scope**

{coherent_access/ access_scope}		Subscriber requests:			
		False/Instance	False/Topic	True/Instance	True/Topic
Publisher offers:	False/Instance	✔	incompatible	incompatible	incompatible
	False/Topic	✔	✔	incompatible	incompatible
	True/Instance	✔	incompatible	✔	incompatible
	True/Topic	✔	✔	✔	✔

6.4.6.5 Related QosPolicies

The DESTINATION_ORDER QosPolicy (Section 6.5.5) is closely related and also affects ordering of data samples on a per-instance basis when there are multiple *DataWriters*.

6.4.6.6 Applicable Entities

- ❑ Publishers (Section 6.2)
- ❑ Subscribers (Section 7.2)

6.4.6.7 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

6.5 DataWriter QosPolicies

This section provides detailed information about the QosPolicies associated with a *DataWriter*. Table 6.15 on page 6-50 provides a quick reference. They are presented in alphabetical order.

- ❏ BATCH QosPolicy (DDS Extension) (Section 6.5.1)
- ❏ DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)
- ❏ DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3)
- ❏ DEADLINE QosPolicy (Section 6.5.4)
- ❏ DESTINATION_ORDER QosPolicy (Section 6.5.5)
- ❏ DURABILITY QosPolicy (Section 6.5.6)
- ❏ DURABILITY SERVICE QosPolicy (Section 6.5.7)
- ❏ HISTORY QosPolicy (Section 6.5.8)
- ❏ LATENCYBUDGET QoS Policy (Section 6.5.9)
- ❏ LIFESPAN QoS Policy (Section 6.5.10)
- ❏ LIVELINESS QosPolicy (Section 6.5.11)
- ❏ MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12)
- ❏ OWNERSHIP QosPolicy (Section 6.5.13)
- ❏ OWNERSHIP_STRENGTH QosPolicy (Section 6.5.14)
- ❏ PROPERTY QosPolicy (DDS Extension) (Section 6.5.15)
- ❏ PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.16)
- ❏ RELIABILITY QosPolicy (Section 6.5.17)
- ❏ RESOURCE_LIMITS QosPolicy (Section 6.5.18)
- ❏ TRANSPORT_PRIORITY QosPolicy (Section 6.5.19)
- ❏ TRANSPORT_SELECTION QosPolicy (DDS Extension) (Section 6.5.20)
- ❏ TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21)
- ❏ TYPESUPPORT QosPolicy (DDS Extension) (Section 6.5.22)
- ❏ USER_DATA QosPolicy (Section 6.5.23)
- ❏ WRITER_DATA_LIFECYCLE QoS Policy (Section 6.5.24)

6.5.1 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) acknowledgement of small samples, in order to increase throughput.

It specifies and configures the mechanism that allows *RTI Data Distribution Service* to collect multiple 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.

This QosPolicy can be used to increase effective throughput dramatically for small data samples. Throughput for small samples (size < 2048 bytes) is typically limited by CPU capacity and not by network bandwidth. Batching many smaller samples to be sent in a single large packet will increase network utilization and thus throughput in terms of samples per second.

It contains the members listed in Table 6.27.

Table 6.27 **DDS_BatchQosPolicy**

Type	Field Name	Description
DDS_Boolean	enable	Enables/disables batching.
DDS_Long	max_data_bytes	Sets the maximum cumulative length of all serialized 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 samples.
DDS_Long	max_samples	Sets the maximum number of samples in a batch. When this limit is reached, the batch is automatically flushed.
struct DDS_Duration_t	max_flush_delay	Sets the maximum flush delay. When this duration is reached, the batch is automatically flushed. The delay is measured from the time the first sample in the batch is written by the application.

Table 6.27 **DDS_BatchQosPolicy**

Type	Field Name	Description
struct DDS_Duration_t	source_timestamp_ resolution	Sets the batch source timestamp resolution. The value of this field determines how the source timestamp is associated with the samples in a batch. A sample written with timestamp 't' inherits the source timestamp 't2' associated with the previous sample, unless ('t' - 't2') is greater than **source_timestamp_resolution**. If **source_timestamp_resolution** is DURATION_INFINITE, every sample in the batch will share the source timestamp associated with the first sample. If **source_timestamp_resolution** is zero, every sample in the batch will contain its own source timestamp corresponding to the moment when the sample was written. The performance of the batching process is better when **source_timestamp_resolution** is set to DURATION_INFINITE.
DDS_Boolean	thread_safe_write	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 samples.

If batching is enabled (not the default), samples are not immediately sent when they are written. Instead, they get collected into a "batch." A batch always contains whole number of samples—a 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 samples are in the batch (**max_samples**).

- A time-limit (**max_flush_delay**) is reached, as measured from the time the first 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.
- The number of 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* ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1).

The **flush()** operation is described in Flushing Batches of Data Samples (Section 6.3.8).

6.5.1.1 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 sample in a coherent set is written, the batch in progress (without including the 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 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 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1) 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 Flow Controllers (DDS Extension) (Section 6.6). 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 sample.

6.5.1.2 Batching vs. Coalescing

Even when batching is disabled, *RTI Data Distribution Service* will sometimes coalesce multiple samples into a single network datagram. For example, samples buffered by a FlowController or sent in response to a negative acknowledgement (NACK) may be coalesced. This behavior is distinct from sample batching.

Samples that are sent individually (not part of a batch) are always treated as separate samples by *RTI Data Distribution Service*. Each sample is accompanied by a complete RTPS header on the network (although 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 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 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 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.

6.5.1.3 Batching and ContentFilteredTopics

When batching is enabled, content filtering is always done on the reader side.

6.5.1.4 Performance Considerations

The purpose of batching is to increase throughput when writing small 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 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 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 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 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 6.29, "DDS_RtpsReliableWriterProtocol_t," on page 6-95.

6.5.1.5 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.

6.5.1.6 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.

<u>All batching configuration occurs on the publishing side.</u> A subscribing application does not configure anything specific to receive batched samples, and in many cases, it will be oblivious to whether the 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 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1) must be FALSE.
- ❏ If **thread_safe_write** is FALSE, **source_timestamp_resolution** must be DURATION_INFINITE.

6.5.1.7 Related QosPolicies

- ❏ To flush batches based on a time limit, enable batching in the ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1) of the *DataWriter's Publisher*.
- ❏ Be careful when configuring a *DataWriter's* LIFESPAN QoS Policy (Section 6.5.10) with a **duration** shorter than the batch flush period (**max_flush_delay**). If the batch does not fill up before the flush period elapses, the short **duration** will cause the samples to be lost without being sent.
- ❏ Do not configure the *DataReader's or DataWriter's* HISTORY QosPolicy (Section 6.5.8) to be shallower than the *DataWriter's* maximum batch size (**max_samples**). When the HISTORY QosPolicy is shallower on the DataWriter, some samples may not be sent. When the HISTORY QosPolicy is shallower on the DataReader, samples may be dropped before being provided to the application.
- ❏ The initial and maximum numbers of batches that a *DataWriter* will manage is set in the DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3).
- ❏ The maximum number of samples that a *DataWriter* can store is determined by the value **max_samples** in the RESOURCE_LIMITS QosPolicy (Section 6.5.18) and **max_batches** in the DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3). The limit that is reached first is applied.
- ❏ The amount of resources required for batching depends on the configuration of the RESOURCE_LIMITS QosPolicy (Section 6.5.18) and the DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3). See Section 6.5.1.9.

6.5.1.8 Applicable Entities

- ❏ DataWriters (Section 6.3)

6.5.1.9 System Resource Considerations

- ❏ Batching requires additional resources to store the meta-data associated with the 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 Section 6.5.1.7.

6.5.2 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension)

DDS has a standard protocol for packet (user and meta data) exchange between applications using DDS for communications. 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 DDS responds to "slow" reliable *DataReaders* or ones that disconnect or are otherwise lost.

This policy includes the members presented in Table 6.28, “DDS_DataWriterProtocolQosPolicy,” on page 6-93 and Table 6.29, “DDS_RtpsReliableWriterProtocol_t,” on page 6-95. For defaults and valid ranges, please refer to the online documentation.

For details on the reliability protocol used by *RTI Data Distribution Service*, see Chapter 10. See the RELIABILITY QosPolicy (Section 6.5.17) for more information on per-*DataReader*/*DataWriter* reliability configuration. The HISTORY QosPolicy (Section 6.5.8) and RESOURCE_LIMITS QosPolicy (Section 6.5.18) also play important roles in the DDS reliability protocol.

Table 6.28 **DDS_DataWriterProtocolQosPolicy**

Type	Field Name	Description
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 *RTI Data Distribution Service* to remember information about a *DataWriter* that may be deleted and then recreated. *RTI Data Distribution Service* uses the virtual GUID to associate a durable writer history to a *DataWriter*. *RTI DDS Persistence Service* uses the virtual GUID to send 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 Durability and Persistence Based on Virtual GUIDs (Section 11.2). By default, *RTI Data Distribution Service* 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**, **rtps_instance_id** in the WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9), together with the 3 least significant bytes in **rtps_object_id**, and another byte assigned by *RTI Data Distribution Service* to identify the entity type, forms the BuiltinTopicKey in PublicationBuiltinTopicData.
DDS_Boolean	push_on_write	Controls when a sample is sent after **write()** is called on a *DataWriter*. If TRUE, the sample is sent immediately; if FALSE, the sample is put in a queue until an ACK/NACK is received from a reliable *DataReader*. **Note: push_on_write must be TRUE for Asynchronous DataWriters.** Otherwise, samples will never be sent.

Table 6.28 **DDS_DataWriterProtocolQosPolicy**

Type	Field Name	Description
DDS_Boolean	disable_positive_acks	Determines whether matching *DataReaders* send positive acknowledgements (ACKs) to the *DataWriter*. When TRUE, the *DataWriter* will keep samples in its queue for ACK-disabled readers for a minimum keep duration (see Section 6.5.2.3). 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 samples. This field only applies to keyed writers. *RTI Data Distribution Service* 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 sample). **Note:** Setting disable_inline_keyhash to TRUE is not compatible with using *RTI Real-Time Connect* or *RTI Recorder*.
DDS_Boolean	serialize_key_with_dispose	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 DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1)) 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.
DDS_RtpsReliableWriterProtocol_t	rtps_reliable_writer	This structure includes the fields in Table 6.29.

Table 6.29 **DDS_RtpsReliableWriterProtocol_t**

Type	Field Name	Description
DDS_Long	low_watermark	Queue levels that control when to switch between the regular and fast heartbeat rates (heartbeat_period and fast_heartbeat_period). See Section 6.5.2.1.
	high_watermark	
DDS_Duration_t	heartbeat_period	Rates at which to sent heartbeats to *DataReaders* with unacknowledged samples. See Section 6.5.2.2 and Section 10.3.4.1.
	fast_heartbeat_period	
	late_joiner_heartbeat_period	
DDS_Long	max_heartbeat_retries	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 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 Section 10.3.4.4.
DDS_Boolean	inactivate_nonprogressing_readers	Allows the *DataWriter* to treat *DataReaders* that send successive non-progressing NACK packets as inactive. See Section 10.3.4.5.

Table 6.29 **DDS_RtpsReliableWriterProtocol_t**

Type	Field Name	Description
DDS_Long	heartbeats_per_max_samples	**For non-multichannel DataWriters:** • If batching is disabled: A piggyback heartbeat will be sent every [max_samples/ heartbeats_per_max_samples] number of samples. heartbeats_per_max_samples must be <= writer_qos.resource_limits.max_samples • If batching is enabled: A piggyback heartbeat will be sent every [max_batches/ heartbeats_per_max_samples] number of samples. heartbeats_per_max_samples must be <= writer_qos.resource_limits.max_batches **For multi-channel DataWriters:** A piggyback heartbeat will be sent on a channel every [max_samples/heartbeats_per_max_samples] number of samples sent of that channel. heartbeats_per_max_samples must be <= writer_qos.resource_limits.max_samples. See Section 16.6.2 for additional details related to the multi-channel DataWriter reliability protocol.
DDS_Duration_t	min_nack_response_delay	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 samples or a heartbeat. This set the value of the minimum delay. See Section 10.3.4.6.
DDS_Duration_t	max_nack_response_delay	Maximum delay to respond to a ACK/NACK. This sets the value of maximum delay between receiving an ACK/NACK and sending repair 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 Section 10.3.4.6.

Table 6.29 **DDS_RtpsReliableWriterProtocol_t**

Type	Field Name	Description
DDS_Duration_t	nack_suppression_duration	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.
DDS_Long	max_bytes_per_nack_ response	Maximum bytes in a repair package. When a reliable *DataWriter* resends samples, the total package size is limited to this value. See Section 10.3.4.3.
DDS_Duration_t	disable_positive_acks_ min_sample_keep_ duration	Minimum duration that a sample will be kept in the *DataWriter's* queue for ACK-disabled *DataReaders*. See Section 6.5.2.3 and Section 10.3.4.7.
	disable_positive_acks_ max_sample_keep_ duration	Maximum duration that a sample will be kept in the *DataWriter's* queue for ACK-disabled readers.
DDS_Boolean	disable_positive_acks_ enable_adaptive_ sample_keep_duration	Enables automatic dynamic adjustment of the 'keep duration' in response to network congestion.
DDS_Long	disable_positive_acks_ increase_sample_ keep_duration_factor	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.
	disable_positive_acks_ decrease_sample_ keep_duration_factor	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.

Table 6.29 **DDS_RtpsReliableWriterProtocol_t**

Type	Field Name	Description
DDS_Long	min_send_window_size max_send_window_size	Minimum and maximum size for the window of outstanding samples. See Configuring the Send Window Size (Section 6.5.2.4).
DDS_Long	send_window_decrease_factor	Scales the current send-window size down by this percentage to decrease the effective send-rate in response to received negative acknowledgement. See Configuring the Send Window Size (Section 6.5.2.4).
DDS_Long	send_window_increase_factor	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 Configuring the Send Window Size (Section 6.5.2.4)
DDS_Duration	send_window_update_period	Period in which *DataWriter* checks for received negative acknowledgements and conditionally increases the send-window size when none are received. See Configuring the Send Window Size (Section 6.5.2.4)

6.5.2.1 High and Low Watermarks

When the number of unacknowledged samples in the queue of a reliable *DataWriter* exceeds high_watermark, the RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7) will be changed appropriately, a listener callback will be triggered, and the *DataWriter* will start heartbeating its matched *DataReaders* at **fast_heartbeat_rate**.

When the number of samples falls below low_watermark, the RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7) will be changed appropriately, a listener callback will be triggered, and the heartbeat rate will return to the "normal" rate (**heartbeat_rate**).

Having both high and low watermarks (instead of one) helps prevent rapid flickering between the rates, which could happen if the number of 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 and the *DataWriter* is not a multi-channel *DataWriter*, **high_watermark** and **low_watermark** refer to batches, not samples. For multi-channel *DataWriters*, **high_watermark** and **low_watermark** always refer to samples (see Section 16.6.2 for additional details related to the multi-channel *DataWriter* reliability protocol).

6.5.2.2 Normal, Fast, and Late-Joiner Heartbeat Periods

The normal **heartbeat_period** is used until the number of samples in the reliable *DataWriter*'s queue exceeds **high_watermark**; then **fast_heartbeat_period** is used. Once the number of samples drops below **low_watermark**, **heartbeat_period** is used again.

- **fast_heartbeat_period** must be <= **heartbeat_period**

Increasing **fast_heartbeat_period** increases the speed of discovery, but results in a larger surge of traffic when the *DataWriter* is waiting for acknowledgments.

Decreasing **heartbeat_period** 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 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 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 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** is used when a reliable *DataReader* joins after a reliable *DataWriter* (with non-volatile Durability) has begun publishing samples. Once the late-joining *DataReader* has received all cached samples, it will be serviced at the same rate as other reliable *DataReaders*.

- **late_joiner_heartbeat_period** must be <= **heartbeat_period**

6.5.2.3 Disabling Positive Acknowledgements

When strict reliable communication is not required, you can configure *RTI Data Distribution Service* 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 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 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)).
- ❏ Disable ACKs for individual *DataReaders* (by setting **disable_positive_acks** to TRUE in the DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1)).

If ACKs are disabled, instead of the *DataWriter* holding a sample in its send queue until all of its *DataReaders* have ACKed it, the *DataWriter* will hold a sample for a configurable duration. This “keep-duration" starts when a sample is written. When this time elapses, the 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_acks_min_sample_keep_duration** and **rtps_reliable_writer.disable_positive_acks_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 sample in its queue until all ACK-enabled *DataReaders* have ACKed it and the "keep-duration" has elapsed.

See also: Disabling Positive Acknowledgements (disable_postive_acks_min_sample_keep_duration) (Section 10.3.4.7).

6.5.2.4 Configuring the Send Window Size

When a reliable *DataWriter* writes a sample, it keeps the sample in its queue until it has received acknowledgements from all of its subscribing *DataReaders*. The number of these outstanding samples is referred to as the *DataWriter's* "send window." Once the number of outstanding samples has reached the send window size, subsequent writes will block until an outstanding sample is acknowledged.

Configuration of the send window sets a minimum and maximum size, which may be unlimited. 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**.

6.5.2.5 Propagating Serialized Keys with Disposed-Instance Notifications

This section describes the interaction between these two fields:

- ❏ **serialize_key_with_dispose** in DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)
- ❏ **propagate_dispose_of_unregistered_instances** in DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1)

RTI recommends setting **serialize_key_with_dispose** 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: Disposing of Data (Section 6.3.11.2).

Example 1

1. *DataWriter's* **serialize_key_with_dispose** = false
2. *DataReader's* **propagate_dispose_of_unregistered_instances** = true
3. *DataWriter* calls **dispose()** before writing any 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_with_dispose** = true
2. *DataReader's* **propagate_dispose_of_unregistered_instances** = false
3. *DataWriter* calls **dispose()** before writing any samples
4. *DataReader* calls **take()**, which does not return any 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_with_dispose** = true
2. *DataReader's* **propagate_dispose_of_unregistered_instances** = true
3. *DataWriter* calls **dispose()** before writing any 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 sample with a key
4. *DataWriter* calls **dispose()**, which writes a disposed-instance notification with a key
5. *DataReader* calls **take()** and receives a data sample and a disposed-instance notification; both have keys
6. *DataReader* calls **get_key_value()** with no errors

6.5.2.6 Example

For information on how to use the fields in Table 6.29, see Controlling Heartbeats and Retries with the DataWriterProtocol QosPolicy (Section 10.3.4).

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 a 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 RESOURCE_LIMITS QosPolicy (Section 6.5.18)), 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, *RTI Data Distribution Service* 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.

6.5.2.7 Properties

This QosPolicy cannot be modified after the *DataWriter* is created.

Since it is only for *DataWriter*s, 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, *RTI Data Distribution Service* returns **DDS_RETCODE_INCONSISTENT_POLICY**:

- **min_nack_response_delay** <= **max_nack_response_delay**
- **fast_heartbeat_period <= heartbeat_period**
- **late_joiner_heartbeat_period** <= **heartbeat_period**
- **low_watermark < high_watermark**
- If batching is disabled or the *DataWriter* is a multi-channel *DataWriter*:
  - **heartbeats_per_max_samples** <= **writer_qos.resource_limits.max_samples**

6. Sending Data

❏ If batching is enabled and the *DataWriter* is not a multi-channel *DataWriter*:

- **heartbeats_per_max_samples** <= **writer_qos.resource_limits.max_batches**

6.5.2.8 Related QosPolicies

❏ DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1)

❏ HISTORY QosPolicy (Section 6.5.8)

❏ RELIABILITY QosPolicy (Section 6.5.17)

6.5.2.9 Applicable Entities

❏ DataWriters (Section 6.3)

6.5.2.10 System Resource Considerations

A high **max_bytes_per_nack_response** may increase the instantaneous network bandwidth required to send a single burst of traffic for resending dropped packets.

6.5.3 DATA_WRITER_RESOURCE_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 6.30. For defaults and valid ranges, please refer to the online documentation.

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 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 *RTI Data Distribution Service* from dynamically allocating any memory after the creation of the *DataWriter*.

Table 6.30 **DDS_DataWriterResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_Long	initial_concurrent_blocking_threads	Initial number of threads that are allowed to concurrently block on the **write()** call on the same *DataWriter*.
	max_concurrent_blocking_threads	Maximum number of threads that are allowed to concurrently block on **write()** call on the same *DataWriter*.
	max_remote_reader_filters	Maximum number of remote *DataReaders* for which this *DataWriter* will perform content-based filtering.
	initial_batches	Initial number of batches that a *DataWriter* will manage if batching is enabled.
	max_batches	Maximum number of batches that a *DataWriter* will manage if batching is enabled. When batching is enabled, the maximum number of samples that a *DataWriter* can store is limited by this value and **max_samples** in RESOURCE_LIMITS QosPolicy (Section 6.5.18).
DDS_DataWriter ResourceLimitsInstanceReplacementKind	instance_replacement	Sets the kinds of instances allowed to be replaced when a *DataWriter* reaches instance resource limits. (see Configuring DataWriter Instance Replacement (Section 6.5.18.2)
DDS_Boolean	replace_empty_instances	Whether to replace empty instances during instance replacement.(see Configuring DataWriter Instance Replacement (Section 6.5.18.2)
DDS_Boolean	autoregister_instances	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.

When setting the fields in this policy, the following rule applies. If this is false, *RTI Data Distribution Service* returns **DDS_RETCODE_INCONSISTENT_POLICY**:

- ❑ **max_concurrent_blocking_threads** >= **initial_concurrent_blocking_threads**

The **initial_concurrent_blocking_threads** is the used to allocate necessary system resource initially. 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 RELIABILITY QosPolicy (Section 6.5.17). This QoS is offered so that the user application can control the dynamic allocation of system resources by *RTI Data Distribution Service*.

If you do not mind if *RTI Data Distribution Service* 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 **write()** on the same *DataWriter*, and you do not want *RTI Data Distribution Service* to allocate any system resources or memory after initialization, then you should set:

max_concurrent_blocking_threads = **initial_concurrent_blocking_threads** = *NUM*

(where *NUM* is the number of threads that could possibly block concurrently).

Each *DataWriter* can perform content-based data filtering for up to **max_remote_reader_filters** number of *DataReaders*. Setting the value to 0 will disable filtering by the *DataWriter*, which means that the *DataReader* will have to filter the data itself. For more information, see ContentFilteredTopics (Section 5.4).

6.5.3.1 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.

6.5.3.2 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.

6.5.3.3 Related QosPolicies

- ❏ BATCH QosPolicy (DDS Extension) (Section 6.5.1)
- ❏ RELIABILITY QosPolicy (Section 6.5.17)
- ❏ HISTORY QosPolicy (Section 6.5.8)

6.5.3.4 Applicable Entities

- ❏ DataWriters (Section 6.3)

6.5.3.5 System Resource Considerations

Increasing the values in this QosPolicy will cause more memory usage and more system resource usage.

6.5.4 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 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 6.31. For the default and valid range, please refer to the online documentation.

Table 6.31 **DDS_DeadlineQosPolicy**

Type	Field Name	Description
DDS_Duration_t	period	For *DataWriters*: maximum time between writing a new value of an instance. For *DataReaders*: maximum time between receiving new values for an instance.

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 sample is received.

RTI Data Distribution Service will modify the **DDS_OFFERED_DEADLINE_MISSED_STATUS** and call the associated method in the **DataWriterListener** (see OFFERED_DEADLINE_MISSED Status (Section 6.3.5.4)) if the application fails to **write()** a value for an instance within the period set by the DEADLINE QosPolicy of the *DataWriter*.

Similarly, *RTI Data Distribution Service* will modify the **DDS_REQUESTED_DEADLINE_MISSED_STATUS** and call the associated method in the *DataReaderListener* (see REQUESTED_DEADLINE_MISSED Status (Section 7.3.6.5)) 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 TIME_BASED_FILTER QosPolicy (Section 7.6.4) may interact such that even though the *DataWriter* writes samples fast enough to fulfill its commitment to its own DEADLINE QosPolicy, the *DataReader* may see violations of *its* DEADLINE QosPolicy. This happens because *RTI Data Distribution Service* 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:

```
reader deadline period >=
                     reader minimum_separation + writer deadline period
```

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 *RTI Data Distribution Service*, see Section 5.1.3.

6.5.4.1 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 else 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 failover to lower strength *DataWriters*, see the OWNERSHIP QosPolicy (Section 6.5.13).

6.5.4.2 Properties

This QosPolicy can be changed at any time.

The deadlines on the two sides must be compatible.

DataWriter's DEADLINE **period** <= the *DataReader*'s DEADLINE **period**.

That is, the *DataReader* cannot expect to receive samples more often than the *DataWriter* commits to sending them.

If the *DataReader* and *DataWriter* have compatible deadlines, *RTI Data Distribution Service* 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.

6.5.4.3 Related QosPolicies

- LIVELINESS QosPolicy (Section 6.5.11)
- OWNERSHIP QosPolicy (Section 6.5.13)
- TIME_BASED_FILTER QosPolicy (Section 7.6.4)

6.5.4.4 Applicable Entities

- Topics (Section 5.1)
- DataWriters (Section 6.3)
- DataReaders (Section 7.3)

6.5.4.5 System Resource Considerations

There is an *RTI Data Distribution Service*-internal thread that 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 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.

6.5.5 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 data sample includes two timestamps: a source timestamp and a destination timestamp. The source timestamp is recorded by the *DataWriter* application when the data was written. The destination timestamp is recorded by the *DataReader* application when the data was received.

This QoS includes the member in Table 6.32.

Each *DataReader* can set this QoS to:

- ❏ **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).

Table 6.32 **DDS_DestinationOrderQosPolicy**

Type	Field Name	Description
DDS_Destination-OrderQosPolicyKind	kind	Can be either: DDS_BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS DDS_BY_SOURCE_TIMESTAMP_ DESTINATIONORDER_QOS
DDS_Duration_t	source_timestamp_tolerance	Allowed tolerance between source timestamps of consecutive samples. Only applies when kind (above) is set to DDS_BY_SOURCE_TIMESTAMP_ DESTINATIONORDER_QOS.

❑ **DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS**

Assuming the OWNERSHIP_STRENGTH allows it, within each instance, the **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 all subscribers will end up with the same final value for the instance.

Data will be delivered by a *DataReader* in the order in which it was sent. If data arrives on the network with a source timestamp earlier than the source timestamp of the last data delivered, the new data will be dropped. This ordering therefore works best when system clocks are relatively synchronized among writing machines.

Not all data sent by multiple *DataWriters* may be delivered to a *DataReader* and not all *DataReaders* will see the same data sent by *DataWriters*. However, all *DataReaders* will see the same "final" data when *DataWriters* "stop" sending data.

- For a *DataWriter* with **kind** DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS: When writing a sample, its timestamp must not be less than the timestamp of the previously written sample. However, if it is less than the timestamp of the previously written sample but the difference is less than this tolerance, the sample will use the previously written sample's timestamp as its timestamp. Otherwise, if the difference is greater than this tolerance, the write will fail.

  See also: Special instructions for deleting DataWriters if you are using the 'Timestamp' APIs and BY_SOURCE_TIMESTAMP Destination Order: on page 6-26.

- A *DataReader* with **kind** DDS_BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS will accept a sample only if the difference between the sample's source timestamp and the reception timestamp is no greater than **source_timestamp_tolerance**. Otherwise, the sample is rejected.

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 *RTI Data Distribution Service*, see Section 5.1.3.

6.5.5.1 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 6.33.

Table 6.33 **Valid Reader/Writer Combinations of DestinationOrder**

Destination Order		DataReader requests:	
		BY_SOURCE	BY_RECEPTION
DataWriter offers:	BY_SOURCE	✔	✔
	BY_RECEPTION	incompatible	✔

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.

6.5.5.2 Related QosPolicies

- ❏ OWNERSHIP QosPolicy (Section 6.5.13)
- ❏ HISTORY QosPolicy (Section 6.5.8)

6.5.5.3 Applicable Entities

- ❏ Topics (Section 5.1)
- ❏ DataWriters (Section 6.3)
- ❏ DataReaders (Section 7.3)

6.5.5.4 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

6.5.6 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, then 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 samples are stored by the *DataWriter* application for *DataReaders* that are found after the samples were initially written. *DataReaders* use this QoS to request 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' data-samples. (Reliable *DataReaders* may wait for these historical samples, see Section 7.3.5.)

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.

Exactly how many samples are stored by the *DataWriter* or requested by the *DataReader* is controlled using the HISTORY QosPolicy (Section 6.5.8).

For more information, please see Chapter 11: Mechanisms for Achieving Information Durability and Persistence.

The possible settings for this QoS are:

- **DDS_VOLATILE_DURABILITY_QOS** *RTI Data Distribution Service* is not required to send and will not deliver any data samples to *DataReaders* that are discovered after the samples were initially published.
- **DDS_TRANSIENT_LOCAL_DURABILITY_QOS** *RTI Data Distribution Service* will store and send previously published samples for delivery to newly discovered *DataReader*s as long as the *DataWriter* entity still exists. For this setting to be effective, you must also set the RELIABILITY QosPolicy (Section 6.5.17) *kind* to Reliable (not Best Effort). The HISTORY QosPolicy (Section 6.5.8) of the *DataReaders/DataWriters* used by *RTI Persistence Service* determines exactly how many samples are saved or delivered by *RTI Persistence Service*.
- **DDS_TRANSIENT_DURABILITY_QOS** *RTI Data Distribution Service* will store previously published samples in memory using *RTI Persistence Service*[1], which will send the stored data to newly discovered *DataReader*s. The HISTORY QosPolicy

1. *RTI Persistence Service* is included with *RTI Data Distribution Service* Professional Edition and Elite Edition.

(Section 6.5.8) of the *DataReaders/DataWriters* used by *RTI Persistence Service* determines exactly how many samples are saved or delivered *RTI Persistence Service*.

- ❑ **DDS_PERSISTENT_DURABILITY_QOS** *RTI Data Distribution Service* will store previously published samples in permanent storage, like a disk, using *RTI Persistence Service*[1], which will send the stored data to newly discovered *DataReader*s. The HISTORY QosPolicy (Section 6.5.8) determines exactly how many samples are saved or delivered.

This QosPolicy includes the members in Table 6.34. For default settings, please refer to the online documentation.

Table 6.34 **DDS_DurabilityQosPolicy**

Type	Field Name	Description
DDS_Durability QosPolicyKind	kind	**DDS_VOLATILE_DURABILITY_QOS**: Do not save or deliver old samples. **DDS_TRANSIENT_LOCAL_DURABILITY_QOS**: Save and deliver old samples if the *DataWriter* still exists. **DDS_TRANSIENT_DURABILITY_QOS**: Save and deliver old samples using a memory-based service. **DDS_PERSISTENCE_DURABILITY_QOS**: Save and deliver old samples using disk-based service.
DDS_Boolean	direct_ communication	Whether or not a TRANSIENT or PERSISTENT *DataReader* should receive samples directly from a TRANSIENT or PERSISTENT *DataWriter.* When TRUE, a TRANSIENT or PERSISTENT *DataReader* will receive samples directly from the original *DataWriter.* The *DataReader* may also receive samples from *RTI Persistence Service*[a] but the duplicates will be filtered by the middleware. When FALSE, a TRANSIENT or PERSISTENT *DataReader* will receive samples only from the *DataWriter* created by *RTI Persistence Service*. This 'relay communication' pattern provides a way to guarantee eventual consistency. See RTI Persistence Service (Section 11.5.1). This field only applies to *DataReaders*.

a. *RTI Persistence Service* is included with *RTI Data Distribution Service* Professional Edition and Elite Edition. See Chapter 11.

6.5.6.1 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 samples immediately after it is enabled.

6.5.6.2 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 6.35.

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 6.35 **Valid Combinations of Durability 'kind'**

		DataReader requests:			
		VOLATILE	TRANSIENT_LOCAL	TRANSIENT	PERSISTENT
DataWriter offers:	VOLATILE	✔	incompatible	incompatible	incompatible
	TRANSIENT_LOCAL	✔	✔	incompatible	incompatible
	TRANSIENT	✔	✔	✔	incompatible
	PERSISTENT	✔	✔	✔	✔

6.5.6.3 Related QosPolicies

- ❏ HISTORY QosPolicy (Section 6.5.8)
- ❏ RELIABILITY QosPolicy (Section 6.5.17)
- ❏ DURABILITY SERVICE QosPolicy (Section 6.5.7)

6.5.6.4 Applicable Entities

- ❏ Topics (Section 5.1)
- ❏ DataWriters (Section 6.3)
- ❏ DataReaders (Section 7.3)

6.5.6.5 System Resource Considerations

Using this policy with a setting other than **VOLATILE** will cause *RTI Data Distribution Service* to use CPU and network bandwidth to send old samples to matching, newly discovered *DataReaders*. The actual amount of resources depends on the total size of data that needs to be sent.

6.5.7 DURABILITY SERVICE QosPolicy

This QosPolicy is only used if the DURABILITY QosPolicy (Section 6.5.6) is PERSISTENT or TRANSIENT *and* you are using *RTI Persistence Service*, which is included with *RTI Data Distribution Service Professional Edition* and *Elite Edition*. *RTI Persistence Service* is used to store and possibly forward the data sent by the *DataWriter* to *DataReaders* who are created after the data was initially sent.

This QosPolicy configures certain parameters of *RTI 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 *RTI Persistence Service*.

Note however, that *RTI Persistence Service* itself may be configured to ignore these values and instead use values from its own configuration file.

For more information, please see:

- ❏ Chapter 11: Mechanisms for Achieving Information Durability and Persistence
- ❏ Chapter 21: Introduction to RTI Persistence Service
- ❏ Chapter 22: Configuring RTI Persistence Service

This QosPolicy includes the members in Table 6.36. For default values, please refer to the online documentation.

Table 6.36 **DDS_DurabilityServiceQosPolicy**

Type	Field Name	Description
DDS_Duration_t	service_cleanup_delay	How long to keep all information regarding an instance.
DDS_HistoryQosPolicyKind	history_kind	Settings to use for the HISTORY QosPolicy (Section 6.5.8) when recouping durable data.
DDS_Long	history_depth	
DDS_Long	max_samples	Settings to use for the RESOURCE_LIMITS QosPolicy (Section 6.5.18) when feeding data to a late joiner.
	max_instances	
	max_samples_per_instance	

The **service_cleanup_delay** in this QosPolicy controls when *RTI Persistence Service* may remove all information regarding a data-instances. 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. While in the NOT_ALIVE_DISPOSED state, *RTI Data Distribution Service* detects that there are no more 'live' *DataWriters* writing the instance. That is, all existing writers either unregister the instance (call unregister) or lose their liveliness.
3. A time interval longer that DurabilityService QosPolicy's **service_cleanup_delay** has elapsed since the time that *RTI Data Distribution Service* detected that the previous two conditions were met.

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 *RTI Data Distribution Service*, see Section 5.1.3.

6.5.7.1 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.

6.5.7.2 Related QosPolicies

- ❏ DURABILITY QosPolicy (Section 6.5.6)
- ❏ HISTORY QosPolicy (Section 6.5.8)
- ❏ RESOURCE_LIMITS QosPolicy (Section 6.5.18)

6.5.7.3 Applicable Entities

- ❏ Topics (Section 5.1)
- ❏ DataWriters (Section 6.3)

6.5.7.4 System Resource Considerations

Since this QosPolicy configures the HISTORY and RESOURCE_LIMITS used by the fictitious *DataReader* and *DataWriter* used by *RTI Persistence Service*, it does have some impact on resource usage.

6.5.8 HISTORY QosPolicy

This QosPolicy configures the number of samples that *RTI Data Distribution Service* will store locally for *DataWriters* and *DataReaders*. For keyed *Topics*, this QosPolicy applies on a per instance basis, so that *RTI Data Distribution Service* will attempt to store the configured value of samples for every instance (see Samples, Instances, and Keys (Section 2.2.2) for a discussion of keys and instances).

It includes the members seen in Table 6.37. For defaults and valid ranges, please refer to the online documentation.

Table 6.37 **DDS_HistoryQosPolicy**

Type	Field Name	Description
DDS_HistoryQos-PolicyKind	kind	DDS_KEEP_LAST_HISTORY_QOS: keep the last **depth** number of samples per instance. DDS_KEEP_ALL_HISTORY_QOS: keep all samples.[a]
DDS_Long	depth	If **kind** = DDS_KEEP_LAST_HISTORY_QOS, this is how many samples to keep per instance.[b] if **kind** = DDS_KEEP_ALL_HISTORY_QOS, this value is ignored.
DDS_RefilterQos-PolicyKind	refilter	Specifies how a *DataWriter* should handle previously written samples for a new *DataReader*. When a new *DataReader* matches a *DataWriter*, the *DataWriter* can be configured to perform content-based filtering on previously written samples stored in the *DataWriter* queue for the new *DataReader*. May be: ❏ DDS_NONE_REFILTER_QOS Do not filter existing samples for a new *DataReader*. The *DataReader* will do the filtering. ❏ DDS_ALL_REFILTER_QOS Filter all existing samples for a newly matched *DataReader*. ❏ DDS_ON_DEMAND_REFILTER_QOS Filter existing samples only when they are requested by the *DataReader*. (An extension to the DDS standard.)

a. *RTI Data Distribution Service* will store up to the value of the **max_samples_per_instance** parameter of the RESOURCE_LIMITS QosPolicy (Section 6.5.18).

b. **depth** must be <= **max_samples_per_instance** parameter of the RESOURCE_LIMITS QosPolicy (Section 6.5.18)

6. Sending Data

The **kind** determines whether or not to save a configured number of samples or *all* samples. It can be set to either of the following:

- ❑ **DDS_KEEP_LAST_HISTORY_QOS** *RTI Data Distribution Service* 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**.
  - For a *DataWriter*: *RTI Data Distribution Service* attempts to keep the most recent **depth** samples of each instance (identified by a unique key) managed by the *DataWriter*.
  - For a *DataReader*: *RTI Data Distribution Service* attempts to keep the most recent **depth** samples received for each instance (identified by a unique key) until the application takes them via the *DataReader*'s **take()** operation. See Section 7.4.3 for a discussion of the difference between **read()** and **take()**.
- ❑ **DDS_KEEP_ALL_HISTORY_QOS** *RTI Data Distribution Service* attempts to keep all of the samples of a *Topic*.
  - For a *DataWriter*: *RTI Data Distribution Service* attempts to keep all samples published by the *DataWriter*.
  - For a *DataReader*: *RTI Data Distribution Service* attempts to keep all 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 Section 7.4.3 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 samples kept is subject to the limitations imposed by the RESOURCE_LIMITS QosPolicy (Section 6.5.18). All of the 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 difference instances for a *Topic*, it is possible that the physical queue may run out of space before the number of samples reaches the **depth** for all instances.

In the **KEEP_ALL** case, *RTI Data Distribution Service* can only keep as many samples for a *Topic* (independent of instances) as the size of the allocated queue. *RTI Data Distribution Service* 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 RELIABILITY QosPolicy (Section 6.5.17) by controlling whether or not 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 RESOURCE_LIMITS QosPolicy (Section 6.5.18). Also, the amount of data that is sent to new *DataReaders* who have configured their DURABILITY QosPolicy (Section 6.5.6) to receive previously published data is controlled by the History QosPolicy.

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 samples for an instance in the queue reaches the value of **depth**, a new sample for the instance will replace the oldest sample for the instance in the queue.
  - If RELIABILITY is **RELIABLE**: When the number of samples for an instance in the queue reaches the value of **depth**, a new sample for the instance will replace the oldest sample for the instance in the queue—even if the sample being overwritten has not been fully acknowledged as being received by all reliable *DataReaders*. This implies that the discarded sample may be lost by some reliable *DataReaders*. Thus, when using the **KEEP_LAST** setting, strict reliability is not guaranteed. See Chapter 10: Reliable Communications for a complete discussion on *RTI Data Distribution Service*'s reliable protocol.
- ❑ **DDS_KEEP_ALL_HISTORY_QOS**
  - If RELIABILITY is **BEST_EFFORT**: If the number of samples for an instance in the queue reaches the value of the RESOURCE_LIMITS QosPolicy (Section 6.5.18)'s **max_samples_per_instance** field, a new sample for the instance will replace the oldest sample for the instance in the queue (regardless of instance).
  - If RELIABILITY is **RELIABLE**: When the number of samples for an instance in the queue reaches the value of the RESOURCE_LIMITS QosPolicy (Section 6.5.18)'s **max_samples_per_instance** field, then:

    a) for a *DataWriter*—a new sample for the instance will replace the oldest sample for the instance in the sending queue—*only* if the sample being overwritten has been fully acknowledged as being received by all reliable *DataReaders*. If the oldest sample for the instance has not been fully acknowledged, the **write()** operation trying to enter a new sample for the instance into the sending queue will block (for the **max_blocking_time** specified in the RELIABLE QosPolicy).

b) for a *DataReader*—a new sample received by the *DataReader* will be discarded. Because the *DataReader* will not acknowledge the discarded sample, the *DataWriter* is forced to resend the sample. Hopefully, the next time the sample is received, there is space for the instance in the *DataReader*'s queue to store (and accept, thus acknowledge) the sample. A sample will remain in the *DataReader*'s queue for one of two reasons. The more common reason is that the user application has not removed the sample using the *DataReader*'s **take()** method. Another reason is that the sample has been received out of order and is not available to be taken or read by the user application until all older samples have been received.

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 *RTI Data Distribution Service*, see Section 5.1.3.

6.5.8.1 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 Chapter 10 for a complete discussion on *RTI Data Distribution Service*'s reliable protocol.

See Controlling Queue Depth with the History QosPolicy (Section 10.3.3).

6.5.8.2 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.

6.5.8.3 Related QosPolicies

- ❏ BATCH QosPolicy (DDS Extension) (Section 6.5.1) 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 samples in it.
- ❏ RELIABILITY QosPolicy (Section 6.5.17)
- ❏ RESOURCE_LIMITS QosPolicy (Section 6.5.18)

6.5.8.4 Applicable Entities

- ❑ Topics (Section 5.1)
- ❑ DataWriters (Section 6.3)
- ❑ DataReaders (Section 7.3)

6.5.8.5 System Resource Considerations

While this QosPolicy does not directly affect the system resources used by *RTI Data Distribution Service*, the RESOURCE_LIMITS QosPolicy (Section 6.5.18) that must be used in conjunction with the HISTORY QosPolicy (Section 6.5.8) will affect the amount of memory that *RTI Data Distribution Service* will allocate for a *DataWriter* or *DataReader*.

6.5.9 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. *RTI Data Distribution Service* uses it to prioritize the sending of asynchronously published data; see ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1).

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 6.38.

Table 6.38 **DDS_LatencyBudgetQosPolicy**

Type	Field Name	Description
DDS_Duration_t	duration	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.

6.5.9.1 Applicable Entities

- ❑ Topics (Section 5.1)
- ❑ DataWriters (Section 6.3)
- ❑ DataReaders (Section 7.3)

6.5.10 LIFESPAN QoS Policy

The purpose of this QoS is to avoid delivering stale data to the application. 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 *DataReader* caches, as well as from the transient and persistent information caches.

The middleware attaches timestamps to all data sent and received. The expiration time of each sample is computed by adding the duration specified by this QoS to the destination timestamp. To avoid inconsistencies, if you have multiple *DataWriters* of the same instance, they should all use the same value for this QoS.

When you specify a finite Lifespan for your data, *RTI Data Distribution Service* will compare the current time with those timestamps and drop data when your specified Lifespan expires.

The Lifespan QosPolicy can be used to control how much data is stored by *RTI Data Distribution Service*. Even if it is configured to store "all" of the data sent or received for a topic (see the HISTORY QosPolicy (Section 6.5.8)), 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 6.39. For default and valid range, please refer to the online documentation.

Table 6.39 **DDS_LifespanQosPolicy**

Type	Field Name	Description
DDS_Duration_t	duration	Maximum duration for the data's validity.

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 *RTI Data Distribution Service*, see Setting Topic QosPolicies (Section 5.1.3).

6.5.10.1 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.

6.5.10.2 Related QoS Policies

- BATCH QosPolicy (DDS Extension) (Section 6.5.1) 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** will cause the samples to be lost without being sent.
- DURABILITY QosPolicy (Section 6.5.6)

6.5.10.3 Applicable Entities

- Topics (Section 5.1)
- DataWriters (Section 6.3)

6.5.10.4 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

6.5.11 LIVELINESS QosPolicy

The LIVELINESS QosPolicy specifies how *RTI Data Distribution Service* determines whether a *DataWriter* is "alive." A *DataWrite*r's liveliness is used in combination with the OWNERSHIP QosPolicy (Section 6.5.13) to maintain ownership of an instance (note that the DEADLINE QosPolicy (Section 6.5.4) 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 6.40. For defaults and valid ranges, please refer to the online documentation.

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*.

Table 6.40 **DDS_LivelinessQosPolicy**

Type	Field Name	Description
DDS_Liveliness QosPolicyKind	kind	DDS_AUTOMATIC_LIVELINESS_QOS: *RTI Data Distribution Service* will automatically assert liveliness for the *DataWriter* at least as often as the lease_duration. DDS_MANUAL_BY_PARTICIPANT_LIVELINESS_QOS: The *DataWriter* is assumed to be alive if any *Entity* within the same *DomainParticipant* has asserted its liveliness. DDS_MANUAL_BY_TOPIC_LIVELINESS_QOS: Your application must explicitly assert the liveliness of the *DataWriter* within the lease_duration.
DDS_Duration_t	lease_duration	The timeout by which liveliness must be asserted for the *DataWriter* or else the *DataWriter* will be considered "inactive or not alive. Additionally, for *DataReaders*, the lease_duration also specifies the maximum period at which *RTI Data Distribution Service* will check to see if the matching *DataWriter* is still alive.

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 least as often as required by the **lease_duration**. 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 *RTI Data Distribution Service* 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** — *RTI Data Distribution Service* 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.

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*. Its 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 *RTI Data Distribution Service* finds that a *DataWriter* has failed to have its liveliness asserted by its **lease_duration**, an internal thread will modify the *DataWriter*'s **DDS_LIVELINESS_LOST_STATUS** and trigger its **on_liveliness_lost()** *DataWriterListener* callback if a listener exists, see Listeners (Section 4.4).

Setting the *DataReader*'s **kind** of LIVELINESS requests a specific mechanism for the publishing application to maintain the liveliness of *DataWriter*s. 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 *RTI Data Distribution Service* 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), *RTI Data Distribution Service* will modify the **DDS_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 *RTI Data Distribution Service*, see Section 5.1.3.

For more information on Liveliness, see Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT (Section 12.3.1.2).

6.5.11.1 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 OWNERSHIP QosPolicy (Section 6.5.13) 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 *RTI Data Distribution Service* will start delivering data from the next highest strength *DataWriter* that is still alive.

6.5.11.2 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, *both* of the following conditions must be true:

1. The *DataWriter* and *DataReader* must use one of the valid combinations shown in Table 6.41.
2. *DataWriter*'s **lease_duration** <= *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.

6.5.11.3 Related QosPolicies

- ❑ DEADLINE QosPolicy (Section 6.5.4)
- ❑ OWNERSHIP QosPolicy (Section 6.5.13)
- ❑ OWNERSHIP_STRENGTH QosPolicy (Section 6.5.14)

Table 6.41 **Valid Combinations of Liveliness 'kind'**

		DataReader requests:		
		MANUAL_BY_ TOPIC	**MANUAL_BY_ PARTICIPANT**	**AUTOMATIC**
DataWriter offers:	**MANUAL_BY_ TOPIC**	✔	✔	✔
	MANUAL_BY_ PARTICIPANT	incompatible	✔	✔
	AUTOMATIC	incompatible	incompatible	✔

6. Sending Data

6.5.11.4 Applicable Entities

- ❑ Topics (Section 5.1)
- ❑ DataWriters (Section 6.3)
- ❑ DataReaders (Section 7.3)

6.5.11.5 System Resource Considerations

An internal thread in *RTI Data Distribution Service* will wake up periodically to check the liveliness 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** set on *DataWriters* may also use more network bandwidth because liveliness packets are being sent at a higher rate—this is especially true when LIVELINESS **kind** is set to **AUTOMATIC**.

6.5.12 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 QoS, 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 Chapter 16 for complete documentation on multi-channel *DataWriters*.

Notes:

- ❑ Durable writer history is not supported for multi-channel *DataWriters* (see Chapter 16); an error is reported if a multi-channel *DataWriter* tries to configure Durable Writer History.
- ❑ Asynchronous publishing (see Section 6.4.1) is not supported for multi-channel *DataWriters*; an error is returned if a multi-channel *DataWriter* is configured for asynchronous publishing.

This policy includes the members presented in Table 6.42, Table 6.43, and Table 6.44. For defaults and valid ranges, please refer to the online documentation.

Table 6.42 **DDS_MultiChannelQosPolicy**

Type	Field Name	Description
DDS_ChannelSettingsSeq	channels	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 6.43.
char *	filter_name	Name of the filter class used to describe the filter expressions. The following values are supported: ❑ DDS_SQLFILTER_NAME[a] (see Section 5.4.6) ❑ DDS_STRINGMATCHFILTER_NAME[a] (see Section 5.4.7)

a. In Java and C#, you can access the names of the builtin filters by using DomainParticipant.SQLFILTER_NAME and DomainParticipant.STRINGMATCHFILTER_NAME.

The format of the **filter_expression** should correspond to one of the following filter classes:

- ❑ DDS_SQLFILTER_NAME (see SQL Filter Expression Notation (Section 5.4.6))
- ❑ DDS_STRINGMATCHFILTER_NAME (see STRINGMATCH Filter Expression Notation (Section 5.4.7)

A *DataReader* can use the ContentFilteredTopic API (see Using a ContentFilteredTopic (Section 5.4.5)) to subscribe to a subset of the channels used by a *DataWriter*.

6.5.12.1 Example

See Chapter 16: Multi-channel DataWriters.

Table 6.43 **DDS_ChannelSettings_t**

Type	Field Name	Description
DDS_MulticastSettingsSeq	multicast_settings	A sequence of multicast settings used to configure the multicast addresses associated with a channel. The sequence cannot be empty. The maximum number of multicast locators in a channel is limited to four. (A locator is defined by a transport alias, a multicast address and a port.) See Table 6.44.
char *	filter_expression	A logical expression used to determine the data that will be published in the channel. This string cannot be NULL. An empty string always evaluates to TRUE. See SQL Filter Expression Notation (Section 5.4.6) and STRINGMATCH Filter Expression Notation (Section 5.4.7) for expression syntax.

Table 6.44 **DDS_MulticastSettings**

Type	Field Name	Description
DDS_StringSeq	transports	A sequence of transport aliases that specifies which transport should be used to publish multicast messages for this channel.
char *	receive_address	A multicast group address on which *DataReaders* subscribing to this channel will receive data.
DDS_Long	receive_port	The multicast port on which *DataReaders* subscribing to this channel will receive data.

6.5.12.2 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.

6.5.12.3 Related Qos Policies

- ❏ DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)

6.5.12.4 Applicable Entities

- ❑ DataWriters (Section 6.3)

6.5.12.5 System Resource Considerations

The following fields in the DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4) 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 Chapter 16: Multi-channel DataWriters.

6.5.13 OWNERSHIP QosPolicy

The OWNERSHIP QosPolicy specifies whether a *DataReader* receive 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 6.45.

Table 6.45 **DDS_OwnershipQosPolicy**

Type	Field Name	Description
DDS_OwnershipQosPolicyKind	kind	DDS_SHARED_OWNERSHIP_QOS or DDS_EXCLUSIVE_OWNERSHIP_QOS

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 *DataWriters*. 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 *RTI Data Distribution Service*, see Section 5.1.3.

6.5.13.1 How RTI Data Distribution Service 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 OWNERSHIP_STRENGTH QosPolicy (Section 6.5.14) that is "alive" as defined by the LIVELINESS QosPolicy (Section 6.5.11)) and has not violated the DEADLINE QosPolicy (Section 6.5.4) of the *DataReader*. OWNERSHIP_STRENGTH is simply an integer set by the *DataWriter*.

As mentioned before, if the *Topic*'s data type is keyed (see Section 2.2.2) then **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, *RTI Data Distribution Service* resolves the tie by choosing the *DataWriter* with the smallest GUID (Globally Unique Identifier, see Section 12.1.1.). 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.

6.5.13.2 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, *RTI Data Distribution Service* 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.

6.5.13.3 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.

6.5.13.4 Related QosPolicies

- ❏ DEADLINE QosPolicy (Section 6.5.4)
- ❏ LIVELINESS QosPolicy (Section 6.5.11)
- ❏ OWNERSHIP_STRENGTH QosPolicy (Section 6.5.14)

6.5.13.5 Applicable Entities

- ❏ Topics (Section 5.1)
- ❏ DataWriters (Section 6.3)
- ❏ DataReaders (Section 7.3)

6.5.13.6 System Resource Considerations

This QosPolicy does not significantly impact the use of system resources.

6.5.14 OWNERSHIP_STRENGTH QosPolicy

The OWNERSHIP_STRENGTH QosPolicy is used to rank *DataWriters* of the same instance of a *Topic*, so that *RTI Data Distribution Service* can decide which *DataWriter* will have ownership of the instance when the OWNERSHIP QosPolicy (Section 6.5.13) is set to **EXCLUSIVE**.

It includes the member in Table 6.46. For the default and valid range, please refer to the online documentation.

Table 6.46 **DDS_OwnershipStrengthQosPolicy**

Type	Field Name	Description
DDS_Long	value	The strength value used to arbitrate among multiple *DataWriters*.

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 Section 6.5.13.1 for more details.

6.5.14.1 Example

Suppose there are two *DataWriters* sending 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*.

6.5.14.2 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.

6.5.14.3 Related QosPolicies

- ❏ OWNERSHIP QosPolicy (Section 6.5.13)

6.5.14.4 Applicable Entities

- ❏ DataWriters (Section 6.3)

6.5.14.5 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

6.5.15 PROPERTY QosPolicy (DDS Extension)

The PROPERTY QosPolicy stores name/value (string) pairs that can be used to configure certain parameters of *RTI Data Distribution Service* 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 builtin topic).

It includes the member in Table 6.47.

Table 6.47 **DDS_PropertyQosPolicy**

Type	Field Name	Description
DDS_PropertySeq	value	A sequence of: (name, value) pairs and booleans that indicate whether the pair should be propagated (included in the entity's builtin topic upon discovery).

The Property QoS stores name/value pairs for a DDS Entity. Both the name and value are strings. Certain configurable parameters for DDS Entities that do not have a formal DDS QoS definition may be configured via this QoS by using a pre-defined name and the desired setting in string form.

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 Section 11.3.2)
- ❏ Durable reader state (see Section 11.4.2)
- ❏ Builtin and extension Transport Plugins (see Section 13.6, Section 20.2, Section 29.2).
- ❏ Automatic registration of builtin types (see Registering Built-in Types (Section 3.2.1))
- ❏ Clock Selection (Section 8.6)

In addition, you can add your own name/value pairs to the Property QoS of a DDS Entity. You may also use this QosPolicy to direct *RTI Data Distribution Service* 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 API.s For example, Durable Reader State, Durable Writer History, Builtin Types, Monotonic Clock.
- ❏ Alternative way to configure builtin transports settings. For example, to use non-default values for the builtin transports without using the PropertyQosPolicy, you would have to create a *DomainParticipant* disabled, change the builtin transport property settings, then enable the *DomainParticipant*. Using the PropertyQosPolicy to configure builtin 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 builtin transport settings through the PropertyQosPolicy instead, XML files can be used.
    - **Note**: 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 builtin 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.

---

1. *RTI Secure WAN Transport* is an optional packages available for separate purchase.

2. *RTI TCP Transport* is included with your *RTI Data Distribution Service* distribution but is not a built-in transport and therefore not enabled by default.

The PropertyQosPolicyHelper operations are described in Table 6.48. For more information, see the online documentation.

Table 6.48 **PropertyQoSPolicyHelper Operations**

Operation	Description
get_number_of_properties	Gets the number of properties in the input policy.
assert_property	Asserts the property identified by name in the input policy. (Either adds it, or replaces an existing one.)
add_property	Adds a new property to the input policy.
lookup_property	Searches for a property in the input policy given its name.
remove_property	Removes a property from the input policy.
get_properties	Retrieves a list of properties whose names match the input prefix.

6.5.15.1 Properties

This QosPolicy can be changed at any time.

There is no requirement that the publishing and subscribing sides use compatible values.

6.5.15.2 Related QosPolicies

❑ DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)

6.5.15.3 Applicable Entities

❑ DataWriters (Section 6.3)

❑ DataReaders (Section 7.3)

❑ DomainParticipants (Section 8.3)

6.5.15.4 System Resource Considerations

The DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4) contains several fields for configuring the resources associated with the properties stored in this QosPolicy.

6.5.16 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 *RTI Data Distribution Service*.

Note: Asynchronous *DataWriters* do not perform sender-side filtering. Any filtering, such as time-based or content-based filtering, takes place on the *DataReader* side.

Each *Publisher* spawns a single asynchronous publishing thread (set in its ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1)) to serve all its asynchronous *DataWriters*.

When data is written asynchronously, a FlowController (Section 6.6), identified by *flow_controller_name*, can be used to shape the network traffic. The flow controller's properties determine when the asynchronous publishing thread is allowed to send data and how much.

The fastest way for *RTI Data Distribution Service* 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 ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1)).

In addition, this QosPolicy can select a FlowController to 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 flow controller 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.

This QosPolicy includes the members in Table 6.49. For the defaults, please refer to the online documentation.

The maximum number of samples that will be coalesced depends on **NDDS_Transport_Property_t::gather_send_buffer_count_max** (each sample requires at least 2-4 gather-send buffers). Performance can be improved by increasing **NDDS_Transport_Property_t::gather_send_buffer_count_max**. Note that the maximum value is operating system dependent.

RTI Data Distribution Service queues samples until they can be sent by the asynchronous publishing thread (as determined by the corresponding FlowController).

Table 6.49 **DDS_PublishModeQosPolicy**

Type	Field Name	Description
DDS_PublishMode QosPolicyKind	kind	Either: DDS_ASYNCHRONOUS_PUBLISHER_MODE_QOS[a] DDS_SYNCHRONOUS_PUBLISHER_MODE_QOS
char*	flow_controller_ name	Name of the associated flow controller. There are three built-in FlowControllers: DDS_DEFAULT_FLOW_CONTROLLER_NAME DDS_FIXED_RATE_FLOW_CONTROLLER_NAME DDS_ON_DEMAND_FLOW_CONTROLLER_NAME You may also create your own FlowControllers. See Flow Controllers (DDS Extension) (Section 6.6).

a. See Note on page 6-93.

The number of samples that will be queued is determined by the HISTORY QosPolicy (Section 6.5.8): when using **KEEP_LAST**, the most recent **depth** samples are kept in the queue.

Once unsent samples are removed from the queue, they are no longer available to the asynchronous publishing thread and will therefore never be sent.

Unless **flow_controller_name** points to one of the built-in FlowControllers, finalizing the DataWriterQos will also free the string pointed to by **flow_controller_name**. Therefore, you should use **DDS_String_dup()** before passing the string to **flow_controller_name**, or reset **flow_controller_name** to NULL before the destructing /finalizing the QoS.

Advantages of Asynchronous Publishing:

Asynchronous publishing may increase latency, but offers the following advantages:

- ❏ The **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 samples for the same destination will be coalesced into a single network packet which reduces bandwidth consumption.

6.5.16.1 Properties

This QosPolicy cannot be modified after the *Publisher* is created.

Since it is only for *DataWriters*, there are no compatibility restrictions for how it is set on the publishing and subscribing sides.

6.5.16.2 Related QosPolicies

- ❏ ASYNCHRONOUS_PUBLISHER QosPolicy (DDS Extension) (Section 6.4.1)
- ❏ HISTORY QosPolicy (Section 6.5.8)

6.5.16.3 Applicable Entities

- ❏ DataWriters (Section 6.3)

6.5.16.4 System Resource Considerations

See Configuring Resource Limits for Asynchronous DataWriters (Section 6.5.18.1).

System resource usage depends on the settings in the corresponding FlowController (see Section 6.6).

6.5.17 RELIABILITY QosPolicy

This RELIABILITY QosPolicy determines whether or not data published by a *DataWriter* will be reliably delivered by *RTI Data Distribution Service* to matching *DataReaders*. The reliability protocol used by *RTI Data Distribution Service* is discussed in Chapter 10: Reliable Communications.

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 *RTI Data Distribution Service* 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 a SAMPLE_LOST Status (Section 7.3.6.7) is generated.

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 *RTI Data Distribution Service* 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.

RTI Data Distribution Service uses a reliability protocol configured and tuned by these QoS policies: HISTORY QosPolicy (Section 6.5.8), DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2), DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1), and RESOURCE_LIMITS QosPolicy (Section 6.5.18).

The Reliable QoS policy is simply a switch to turn on the reliability protocol for a *DataWriter/DataReader* connection. The level of reliability provided by *RTI Data Distribution Service* is determined by the configuration of the aforementioned QoS policies.

You can configure *RTI Data Distribution Service* 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). In 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 6.50. For defaults and valid ranges, please refer to the online documentation.

Table 6.50 **DDS_ReliabilityQosPolicy**

Type	Field Name	Description
DDS_ReliabilityQosPolicyKind	kind	DDS_BEST_EFFORT_RELIABILITIY_QOS: Data samples are sent once and missed samples are acceptable. DDS_RELIABLE_RELIABILITY_QOS: *RTI Data Distribution Service* will make sure that data sent is received and missed samples are resent.
DDS_Duration_t	max_blocking_time	How long a *DataWriter* can block on a write() when the send queue is full due to unacknowledged messages. (Has no meaning for *DataReaders*.)

The **kind** of RELIABILITY can be either:

- ❑ **BEST_EFFORT** *RTI Data Distribution Service* will send data samples only once to *DataReaders*. No effort or resources are spent to track whether or not sent 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. Data samples may be lost. This setting is good for periodic data.
- ❑ **RELIABLE** *RTI Data Distribution Service* will send samples reliably to *DataReaders*–buffering sent data until they have been acknowledged as being received by *DataReaders* and resending any 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 samples by the *DataReader*. This setting is a good choice when guaranteed data delivery is required; for example, sending events or commands.

While a *DataWriter* sends data reliably, the HISTORY QosPolicy (Section 6.5.8) and RESOURCE_LIMITS QosPolicy (Section 6.5.18) determine how many samples can be stored while waiting for acknowledgements from *DataReaders*. A sample that is sent reliably is entered in the *DataWriter*'s send queue awaiting acknowledgement from *DataReaders*. How many 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 samples per instance of the *Topic* in the send queue. Should the number of unacknowledge samples in the send queue for a data-instance reach the HISTORY **depth**, then the next sample written by the *DataWriter* for the instance will overwrite the oldest sample for the instance in the queue. This implies that an unacknowledged 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* 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 acknowledged samples (either due to the number of unacknowledged samples for an instance reaching the RESOURCE_LIMITS **max_samples_per_instance** value or the total number of unacknowledged 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 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 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 *RTI Data Distribution Service*, see Section 5.1.3.

6.5.17.1 Example

This QosPolicy is used to achieve reliable communications, which is discussed in Chapter 10: Reliable Communications and Section 10.3.1.

6.5.17.2 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 shown in Table 6.41.

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*.

Table 6.51 **Valid Combinations of Reliability 'kind'**

		DataReader requests:	
		BEST_EFFORT	**RELIABLE**
DataWriter offers:	**BEST_EFFORT**	✔	incompatible
	RELIABLE	✔	✔

6.5.17.3 Related QosPolicies

- ❑ HISTORY QosPolicy (Section 6.5.8)
- ❑ RESOURCE_LIMITS QosPolicy (Section 6.5.18)

6.5.17.4 Applicable Entities

- ❑ Topics (Section 5.1)
- ❑ DataWriters (Section 6.3)
- ❑ DataReaders (Section 7.3)

6.5.17.5 System Resource Considerations

Setting the **kind** to **RELIABLE** will cause *RTI Data Distribution Service* 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 Chapter 10: Reliable Communications).

Setting **max_blocking_time** to a non-zero number may block the sending thread when the RELIABILITY kind is **RELIABLE**.

6.5.18 RESOURCE_LIMITS QosPolicy

For the reliability protocol (and the DURABILITY QosPolicy (Section 6.5.6)), this QosPolicy determines the actual maximum queue size when the HISTORY QosPolicy (Section 6.5.8) is set to KEEP_ALL.

In general, this QosPolicy is used to limit the amount of system memory that 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.

This QosPolicy can be set such that an entity does not dynamically allocate any more memory after its initialization phase.

It includes the members in Table 6.52. For defaults and valid ranges, please refer to the online documentation.

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 Tuning Queue Sizes and Other Resource Limits (Section 10.3.2).

Table 6.52 **DDS_ResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_Long	max_samples	Maximum number of samples that *RTI Data Distribution Service* can store for a *DataWriter/DataReader.* This is a physical limit.
DDS_Long	max_instances	Maximum number of instances that can be managed by a *DataWriter/DataReader.* For *DataReaders*, **max_instances** must be <= **max_total_instances** in the DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2). See also: Example (Section 6.5.18.3).
DDS_Long	max_samples_per_instance	Maximum number of samples of any one instance that *RTI Data Distribution Service* will store for a *DataWriter/DataReader.* If a keyed *Topic* is not used, then **max_samples_per_instance** must equal **max_samples**.
DDS_Long	initial_samples	Initial number of samples that *RTI Data Distribution Service* will store for a *DataWriter/DataReader.* (DDS extension)
DDS_Long	initial_instances	Initial number of instances that can be managed by a *DataWriter/DataReader.* (DDS extension)
DDS_Long	instance_hash_buckets	Number of hash buckets, which are used by *RTI Data Distribution Service* to facilitate instance lookup. (DDS extension).

When a *DataWriter* or *DataReader* 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 samples or as more instances are created, then *RTI Data Distribution Service* 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 *RTI Data Distribution Service* to dynamically allocate memory after initialization.

For keyed *Topics*, the **max_samples_per_instance** field in this policy represents maximum number of samples with the same key that are allowed to be stored by a *DataWriter* or *DataReader.* This 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 *RTI Data Distribution Service* to allocate the total memory needed to hold the maximum number of 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, *RTI Data Distribution Service* must be able to store **max_samples_per_instance** for at least one instance. Therefore, **max_samples_per_instance** must be <= **max_samples**.

Important: If a keyed data type is not used, then 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 *RTI Data Distribution Service* deals with new data samples being sent or received for a *DataWriter* or *DataReader* is described in the HISTORY QosPolicy (Section 6.5.8) 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 *RTI Data Distribution Service*, see Section 5.1.3.

6.5.18.1 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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4) is always greater than the sum of all asynchronous *DataWriters*' **max_samples**.

6.5.18.2 Configuring DataWriter Instance Replacement

When the **max_instances** limit is reached, a *DataWriter* will try to make space for a new instance by replacing an existing instance according to the instance replacement kind set in **instance_replacement**. 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 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 *DataWriter* will not update any further, unregistered instances are replaced before any other instance kinds. This applies for all **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 registering, 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 DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3)) configures whether instances with no 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.

6.5.18.3 Example

If you want to be able to store **max_samples_per_instance** for every instance, then you should set

max_samples >= max_instances * 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**:

max_samples >= max_samples_per_instance

If the HISTORY QosPolicy (Section 6.5.8)'s **kind** is set to **KEEP_LAST**, then you should set:

max_samples_per_instance = HISTORY.depth

6.5.18.4 Properties

This QosPolicy cannot be modified after the Entity is enabled.

There are no requirements that the publishing and subscribing sides use compatible values.

6.5.18.5 Related QosPolicies

- HISTORY QosPolicy (Section 6.5.8)
- RELIABILITY QosPolicy (Section 6.5.17)
- For *DataReaders*, **max_instances** must be <= **max_total_instances** in the DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2)

6.5.18.6 Applicable Entities

- Topics (Section 5.1)
- DataWriters (Section 6.3)
- DataReaders (Section 7.3)

6.5.18.7 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.

tIn addition to the previous encapsulation IDs, *RTI Data Distribution Service* provides two aliases:

- DDS_ENCAPSULATION_CDR_NATIVE refers to DDS_ENCAPSULATION_ID_CDR_BE in big-endian architectures and DDS_ENCAPSULATION_ID_CDR_LE in little-endian architectures.
- DDS_ENCAPSULATION_PL_CDR_NATIVE refers to DDS_ENCAPSULATION_ID_PL_CDR_BE in big-endian architectures and DDS_ENCAPSULATION_ID_PL_CDR_LE in little-endian architectures.

6.5.19 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* basis that the data sent by a *DataWriter* is of a different priority.

It is not specified by DDS 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 a end-user controllable level of data packet priority.

In *RTI Data Distribution Service*, 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*. 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 from a *DataWriter* to a *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 6.53. For the default and valid range, please refer to the online documentation.

Table 6.53 **DDS_TransportPriorityQosPolicy**

Type	Field Name	Description
DDS_Long	value	Hint as to how to set the priority.

RTI Data Distribution Service will propagate the **value** set on a per-*DataWriter* 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.

Although you can set the TRANSPORT_PRIORITY QosPolicy on *Topics*, its value can only be used to initialize the TRANSPORT_PRIORITY QosPolicies of a *DataWriter*. It does not directly affect the operation of *RTI Data Distribution Service*, see Section 5.1.3.

6.5.19.1 Example

Should *RTI Data Distribution Service* 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 data samples, when published, will be sent at a higher priority than other *DataWriters* that use the same transport.

6.5.19.2 Properties

This QosPolicy may be modified after the entity is created.

It does not apply to *DataReaders*, so there is no requirement that the publishing and subscribing sides use compatible values.

6.5.19.3 Related QosPolicies

This QosPolicy does not interact with any other policies.

6.5.19.4 Applicable Entities

- ❏ Topics (Section 5.1)
- ❏ DataWriters (Section 6.3)

6.5.19.5 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.

6.5.20 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 4 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 6.54. For more information, please refer to the online documentation.

Table 6.54 **DDS_TransportSelectionQosPolicy**

Type	Field Name	Description
DDS_StringSeq	enabled_transports	A sequence of aliases for the transports that may be used by the *DataWriter* or *DataReader*.

RTI Data Distribution Service allows user 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 the user 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 online documentation (from the **Modules** page, select **RTI Data Distribution Service API Reference, Pluggable Transports**).

6.5.20.1 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.

6.5.20.2 Properties

This QosPolicy cannot be modified after the *Entity* is created.

It can be set differently for the *DataWriter* and the *DataReader*.

6.5.20.3 Related QosPolicies

- ❑ TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21)
- ❑ TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5)
- ❑ TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)

6.5.20.4 Applicable Entities

- ❑ DataWriters (Section 6.3)
- ❑ DataReaders (Section 7.3)

6.5.20.5 System Resource Considerations

By restricting *DataWriters* from sending or *DataReaders* from receiving over certain transports, you may decrease the load on those transports.

6.5.21 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.

DDS may send data to a variety of DDS Entities, not just *DataReaders*. *DomainParticipants* receive messages to support the discovery process discussed in Chapter 12. *DataWriters* may receive ACK/NACK messages to support the reliable protocol discussed in Chapter 10.

During discovery, each Entity announces to remote applications a list of (up to 4) unicast addresses to which the remote application should use 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 TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8) and TRANSPORT_SELECTION QosPolicy (DDS Extension) (Section 6.5.20)). Also, the associated ports are automatically determined (see Inbound Ports for User Traffic (Section 12.5.2)).

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 4 addresses will be used. 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 6.55. For more information and default values, please refer to the online documentation.

Table 6.55 **DDS_TransportUnicastQosPolicy**

Type	Field Name	Description
DDS_TransportUnicast SettingsSeq (see Table 6.56)	value	A sequence of up to 4 unicast settings that should be used by remote entities to address messages to be sent to this *Entity*.

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

Table 6.56 **DDS_TransportUnicastSettings_t**

Type	Field Name	Description
DDS_StringSeq	transports	A sequence of transport aliases that specifies which transports should be used to receive unicast messages for this *Entity*.
DDS_Long	receive_port	The port that should be used in the addressing of unicast messages destined for this *Entity*. A value of 0 will cause *RTI Data Distribution Service* to use a default port number based on domain and participant ids. See Ports Used for Discovery (Section 12.5).

when sending messages to the Entity on which the TRANSPORT_UNICAST QosPolicy is set.

Up to four "return" unicast addresses may be configured for an *Entity*. 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 online documentation about configuring transports and aliases (from the **Modules** page, select **RTI Data Distribution Service 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 four return addresses that *RTI Data Distribution Service* currently uses.

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 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 online documentation for the builtin 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 *RTI Data Distribution Service* to automatically use a default value. As explained in Ports Used for Discovery (Section 12.5), 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, *RTI Data Distribution Service* may create a thread to process messages received for that port on that transport. See Chapter 17: RTI Data Distribution Service Threading Model 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 MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12), the unicast addresses specified in the TRANSPORT_UNICAST QosPolicy are ignored by that *DataWriter*. The *DataWriter* will not publish samples on those locators.

6.5.21.1 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.

6.5.21.2 Properties

This QosPolicy cannot be modified after the Entity is created.

It can be set differently for the *DomainParticipant*, the *DataWriter* and the *DataReader*.

6.5.21.3 Related QosPolicies

- ❏ MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12)
- ❏ TRANSPORT_SELECTION QosPolicy (DDS Extension) (Section 6.5.20)
- ❏ TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5)
- ❏ TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)

6.5.21.4 Applicable Entities

- ❏ DomainParticipants (Section 8.3)
- ❏ DataWriters (Section 6.3)
- ❏ DataReaders (Section 7.3)

6.5.21.5 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, *RTI Data Distribution Service* 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 Receive Threads (Section 17.3) for more information.

6.5.22 TYPESUPPORT QosPolicy (DDS Extension)

The TypeSupport QosPolicy allows you to set a `void *` value with a *DataWriter* or *DataReader*. This `void *` value is passed to the serialization routine of the data type associated with the *DataWriter* or the deserialization routine of the data type associated with the *DataReader*.

The de/serialization routines may be coded to act differently depending on the information passed in via the `void *` value. The auto-generated serialization and deserialization code by the *rtiddsgen* utility does not use the `void *` value. Usually, for the TypeSupport QosPolicy to have any effect, you will need to use a data type whose serialization/deserialization code has been customized.

This functionality can be used to change how data sent by a *DataWriter* or received by a *DataReader* is serialized or deserialized on a per *DataWriter* and *DataReader* basis. It can also be used to change dynamically how serialization (or for a less common case, deserialization) occurs. For example, a data type could represent a table, including the names of the rows and columns. However, since the row/column names of an instance of the table (a Topic) don't change, they only need to be sent once. The information passed in through the TypeSupport QosPolicy could be used to signal the serialization routine to send the row/column names the first time a *DataWriter* calls **write()**, and then never again.

It includes the members in Table 6.57.

Table 6.57 **DDS_TypeSupportQosPolicy**

Type	Field Name	Description
void *	plugin_data	Value to pass into the type plug-in's serialization/deserialization function.

6.5.22.1 Properties

This QoS policy may be modified after the *DataWriter* or *DataReader* is enabled.

It can be set differently for the *DataWriter* and *DataReader*.

6.5.22.2 Related QoS Policies

None.

6.5.22.3 Applicable Entities

- ❑ DataWriters (Section 6.3)
- ❑ DataReaders (Section 7.3)
- ❑ DomainParticipants (Section 8.3)

6.5.22.4 System Resource Considerations

None.

6.5.23 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 Chapter 12: Discovery) using built-in-topics (see Chapter 14: Built-In Topics). How this information is used will be up to user code. *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 Chapter 14: Built-In Topics).

The structure for the USER_DATA QosPolicy includes just one field, as seen in Table 6.58. 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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4).

Table 6.58 **DDS_UserDataQosPolicy**

Type	Field Name	Description
DDS_OctetSeq	value	Default: empty

This policy is similar to the GROUP_DATA QosPolicy (Section 6.4.4) and TOPIC_DATA QosPolicy (Section 5.2.1) that apply to other types of Entities.

6.5.23.1 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 *RTI Data Distribution Service* 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 Chapter 14: Built-In Topics). 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 *RTI Data Distribution Service* to communicate. See Figure 14.2 for an example of how to do this.

6.5.23.2 Properties

This QosPolicy can be modified at any time. A change in the QosPolicy will cause *RTI Data Distribution Service* to send packets containing the new USER_DATA to all of the other applications in the domain.

It can be set differently on the publishing and subscribing sides.

6.5.23.3 Related QosPolicies

- ❏ TOPIC_DATA QosPolicy (Section 5.2.1)
- ❏ GROUP_DATA QosPolicy (Section 6.4.4)
- ❏ DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)

6.5.23.4 Applicable Entities

- ❏ DataWriters (Section 6.3)
- ❏ DataReaders (Section 7.3)
- ❏ DomainParticipants (Section 8.3)

6.5.23.5 System Resource Considerations

As mentioned earlier, 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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4). Because *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 (Section 6.4.4) and TOPIC_DATA (Section 5.2.1) QosPolicies.

6.5.24 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.

You may use the *DataWriter's* **unregister()** operation to indicate that the *DataWriter* no longer wants to send data for a Topic. This QoS controls whether or not DDS automatically also calls **dispose()** on the behalf of the *DataWriter* for the data.

The behavior controlled by this QoS applies on a per instance (key) basis for keyed Topics, so that when a *DataWriter* unregisters an instance, *RTI Data Distribution Service* can automatically also dispose that instance. This is the default behavior.

In many cases where the ownership of a Topic is shared (see the OWNERSHIP QosPolicy (Section 6.5.13)), *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 regardless of Ownership Strength. In that case, you may only want a *DataWriter* to unregister an instance without disposing the instance. Disposing 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.

When a *DataWriter* unregisters an instance, it means that this particular *DataWriter* has no more information/data on this instance. When an instance is disposed, it means that the instance is "dead"—there will no more information/data from any *DataWriter* on this instance.

This provides the same behavior as explicitly calling one of the **dispose()** operations (Section 6.3.11.2) on the instance before calling **unregister()** (Section 6.3.11.1), provided that **autodispose_unregistered_instances** is set to RTI_TRUE (the default).

When you delete a *DataWriter* (Section 6.3.1), all of the instances managed by the *DataWriter* are automatically unregistered. Therefore, this QoS policy will determine whether or not instances are disposed when the *DataWriter* is deleted by calling one of these operations:

- ❑ *Publisher*'s **delete_datawriter()** (see Section 6.3.1)
- ❑ *Publisher*'s **delete_contained_entities()** (see Section 6.2.3.1**)**
- ❑ *DomainParticipant*'s **delete_contained_entities()** (see Section 8.3.3)

It includes the members in Table 6.59.

Table 6.59 **DDS_WriterDataLifecycleQosPolicy**

Type	Field Name	Description
RTIBool	autodispose_unregistered_instances	RTI_TRUE (default): instance is disposed when unregistered. RTI_FALSE: instance is not disposed when unregistered.

6.5.24.1 Properties

It 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.

6.5.24.2 Related QoS Policies

None.

6.5.24.3 Applicable Entities

❏ DataWriters (Section 6.3)

6.5.24.4 System Resource Considerations

None.

6.6 Flow Controllers (DDS Extension)

A FlowController is the object responsible for shaping the network traffic by determining when attached asynchronous *DataWriters* are allowed to write data.

To use a FlowController, you provide its name in the *DataWriter's* PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.16).

You can use one of these built-in FlowControllers and optionally modify their properties, or create a custom FlowController using the *DomainParticipant's* **create_flowcontroller()** operation (see Section 6.6.4):

❏ 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 built-in DDS_FIXED_RATE_FLOW_CONTROLLER_NAME flow controller shapes the network traffic by allowing data to be sent only once every second. Any accumulated samples destined for the same destination are coalesced into as few network packets as possible.

❏ DDS_ON_DEMAND_FLOW_CONTROLLER_NAME

The built-in DDS_ON_DEMAND_FLOW_CONTROLLER_NAME 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 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 samples (e.g., with the number of samples based on NDDS_Transport_Property_t's **gather_send_buffer_count_max**).

The default property settings for the built-in FlowControllers are listed in the online documentation.

Samples written by an asynchronous *DataWriter* are not sent in the context of the **write()** call. Instead, *RTI Data Distribution Service* puts the samples in a queue for future processing. The FlowController associated with each asynchronous *DataWriter* determines when the samples are actually sent.

Each FlowController maintains a separate FIFO queue for each unique destination (remote application). Samples written by asynchronous *DataWriters* associated with the FlowController are placed in the queues that correspond to the intended destinations of the 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 6.60). Once a queue has been granted tokens, it is serviced by the asynchronous publishing thread. The queued up samples will be coalesced and sent to the corresponding destination. The number of samples sent depends on the data size and the number of tokens granted.

Table 6.60 lists the properties for a FlowController.

Table 6.60 **DDS_FlowControllerProperty_t**

Type	Field Name	Description
DDS_FlowControllerSchedulingPolicy	scheduling_ policy	Either "round-robin" or "earliest-deadline-first." See Section 6.6.1.
DDS_FlowControllerTokenBucketProperty_t	token_bucket	See Section 6.6.3.

Table 6.61 lists the operations available for a FlowController.

6.6.1 Flow Controller Scheduling Policies

- ❑ **Round Robin** (DDS_RR_FLOW_CONTROLLER_SCHED_POLICY) Indicates to flow control in a round-robin (RR) fashion.

Table 6.61 **FlowController Operations**

Operation	Description	Reference
get_property	Get and Set the FlowController properties.	Section 6.6.6
set_property		
trigger_flow	Provides an external trigger to the FlowController.	Section 6.6.7
get_name	Returns the name of the FlowController.	Section 6.6.8
get_participant	Returns the *DomainParticipant* to which the FlowController belongs.	

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) Indicates to flow control in an earliest-deadline-first (EDF) fashion.

  A 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 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 sample with the earliest deadline is serviced first. The number of tokens granted equals the number of tokens required to send the first sample in the queue. Note that the priority of a queue may change as samples are sent (i.e., removed from the queue). If a sample must be sent to multiple destinations or two samples have an equal deadline value, the corresponding destination queues are serviced in a round-robin fashion.

  So 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.

  Since the LatencyBudgetQosPolicy is mutable, a sample written second may contain an earlier deadline than the sample written first if the DDS_LatencyBudgetQosPolicy's **duration** is sufficiently decreased in between writing the two samples. In that case, if the first sample is not yet written (still in queue waiting for its turn), it inherits the priority corresponding to the (earlier) deadline from the second sample.

In other words, the priority of a destination queue is always determined by the earliest deadline among all 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.

6.6.2 Managing Fast DataWriters When Using a FlowController

If a *DataWriter* is writing samples faster than its attached FlowController can throttle, *RTI Data Distribution Service* may drop samples on the writer's side. This happens because the 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 samples are dropped. Then in steady state, the FlowController will smooth out these bursts and the queue will ideally have only one entry.

6.6.3 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 6.62; see the online documentation for their defaults and valid ranges.

Table 6.62 **DDS_FlowControllerTokenBucketProperty_t**

Type	Field Name	Description
DDS_Long	max_tokens	Maximum number of tokens than can accumulate in the token bucket. See Section 6.6.3.1.
DDS_Long	tokens_added_per_period	The number of tokens added to the token bucket per specified period. See Section 6.6.3.2.
DDS_Long	tokens_leaked_per_period	The number of tokens removed from the token bucket per specified period. See Section 6.6.3.3.

Table 6.62 **DDS_FlowControllerTokenBucketProperty_t**

Type	Field Name	Description
DDS_Duration_t	period	Period for adding tokens to and removing tokens from the bucket. See Section 6.6.3.4.
DDS_Long	bytes_per_token	Maximum number of bytes allowed to send for each token available. See Section 6.6.3.5.

Asynchronously published 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. Samples can be sent only when there are sufficient tokens in the bucket. As 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 sample must be sent to multiple destinations, separate tokens are required for each destination. Only when multiple 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.

6.6.3.1 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).

6.6.3.2 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**.

6.6.3.3 tokens_leaked_per_period

When tokens are replenished and there are sufficient tokens to send all 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 samples have been sent. In other words, no token accumulation is allowed. When new samples are written after tokens were purged, the earliest point in time at which they can be sent is at the next periodic replenishment.

6.6.3.4 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).

Note: Once **period** is set to DDS_DURATION_INFINITE, it can no longer be reverted to a finite period.

6.6.3.5 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 sample size, multiple 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 largest message sizes across all transports installed with the *DataWriter*, whichever is less.

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 samples to a single destination. A separate token is required to send to each additional destination.

6.6.4 Creating and Deleting FlowControllers

If you do not want to use one of the three builtin FlowControllers described in Flow Controllers (DDS Extension) (Section 6.6), you can create your own with the *DomainParticipant's* **create_flowcontroller()** operation:

```
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 PUBLISH_MODE QosPolicy (DDS Extension) (Section 6.5.16) (**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 flow controller while another thread may be simultaneously calling **set_default_flowcontroller_property()** or looking for that flow controller with **lookup_flowcontroller()**.

To delete an existing FlowController, use the *DomainParticipant's* **delete_flowcontroller()** operation:

```
DDS_ReturnCode_t delete_flowcontroller (DDSFlowController *  fc)
```

The FlowController must belong this the *DomainParticipant* and not have any attached *DataWriters* or the delete call will return an error (PRECONDITION_NOT_MET).

6. Sending Data

6.6.5 Getting and Setting Default FlowController Properties for DomainParticipants

To get the default DDS_FlowControllerProperty_t values, use this operation on the *DomainParticipant*:

```
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 else, 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*:

```
DDS_ReturnCode_t set_default_flowcontroller_property
                 (const DDS_FlowControllerProperty_t &  property)
```

The special value DDS_FLOW_CONTROLLER_PROPERTY_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_CONTROLLER_PROPERTY_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().**

6.6.6 Getting and Setting Properties for a Specific FlowController

To get the properties of a FlowController, use the FlowController's **get_property()** operation:

```
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:

```
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_CONTROLLER_PROPERTY_DEFAULT can be used to match the current default properties set in the *DomainParticipant*.

6.6.7 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).

```
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).

6.6.8 Other FlowController Operations

If you have the FlowController object and need its name, call the FlowController's **get_name()** operation:

```
const char* DDSFlowController::get_name( )
```

Conversely, if you have the name of the FlowController and need the FlowController object, call the *DomainPartipant's* **lookup_flowcontroller()** operation:

```
DDSFlowController* lookup_flowcontroller (const char *  name)
```

To get a FlowController's *DomainParticipant*, call the FlowController's **get_participant()** operation:

```
DDSDomainParticipant* get_participant ( )
```

Note: It is not safe to lookup a flow controller description while another thread is creating that flow controller.

Chapter 7 Receiving Data

This chapter 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.

This chapter includes the following sections:

- ❏ Preview: Steps to Receiving Data (Section 7.1)
- ❏ Subscribers (Section 7.2)
- ❏ DataReaders (Section 7.3)
- ❏ Using DataReaders to Access Data (Read & Take) (Section 7.4)
- ❏ Subscriber QosPolicies (Section 7.5)
- ❏ DataReader QosPolicies (Section 7.6)

The goal of this chapter 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 online documentation.

7.1 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()** method. This method is also known as polling for data.
- ❏ Your application can be notified asynchronously whenever new data samples arrive—this is done with a *Listener* on either the *Subscriber* or the *DataReader*. *RTI Data Distribution Service* 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()**. *RTI Data Distribution Service* will block your application's thread until the criteria (such as the arrival of 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 Section 7.4 for details on using *DataReaders* to access received data.

See Section 4.6 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*.

1. Type registration is not required for built-in types (see Section 3.2.1).
2. You are not required to explicitly create a *Subscriber*; instead, you can use the 'implicit *Subscriber*' created from the *DomainParticipant*. See Creating Subscribers Explicitly vs. Implicitly (Section 7.2.1).

6. Use a type-safe method to cast the generic *DataReader* created by the *Subscriber* to a type-specific *DataReader*. For example, '**FooDataReader**'.

Now use one of the following mechanisms to receive data.

To receive data samples by polling for new data:

❑ Using a **FooDataReader**, use the **read()** or **take()** operations to access the 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 data samples asynchronously:

❑ Install a *Listener* on the *DataReader* or *Subscriber* that will be called back by an internal *RTI Data Distribution Service* thread when new 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.

   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*.

   RTI Data Distribution Service 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 *RTI Data Distribution Service* 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 data samples.

To wait (block) until data samples arrive:

1. Use the *DataReader* to create a *ReadCondition* that describes the samples for which you want to wait. For example, you can specify that you want to wait for never-before-seen 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 samples. When **wait()** returns, it will indicate that it timed out, or that the attached *Condition* become true (and therefore the desired samples are available).
5. Using a **FooDataReader**, use the **read()** or **take()** operations to access the data samples that have been received and stored for the *DataReader*.

7.2 Subscribers

An application that intends to subscribe to information needs the following DDS Entities: *DomainParticipant*, *Topic*, *Subscriber*, and *DataReader*. All DDS Entities have a corresponding specialized *Listener* and a set of QosPolicies. The *Listener* is how *RTI Data Distribution Service* notifies your application of status changes relevant to the DDS Entity. The QosPolicies allow your application to configure the behavior and resources of the DDS Entity.

- The *DomainParticipant* defines the 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 *DataWriter*'s **read()** or **take()** operation to access 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 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 Creating Subscribers Explicitly vs. Implicitly (Section 7.2.1).

The UML diagram in Figure 7.1 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 7.1. For details such as formal parameters and return codes, please see the online documentation. Otherwise, you can find more information about the operations by looking in the section listed under the Reference column.

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

Figure 7.1 **Subscription Module**

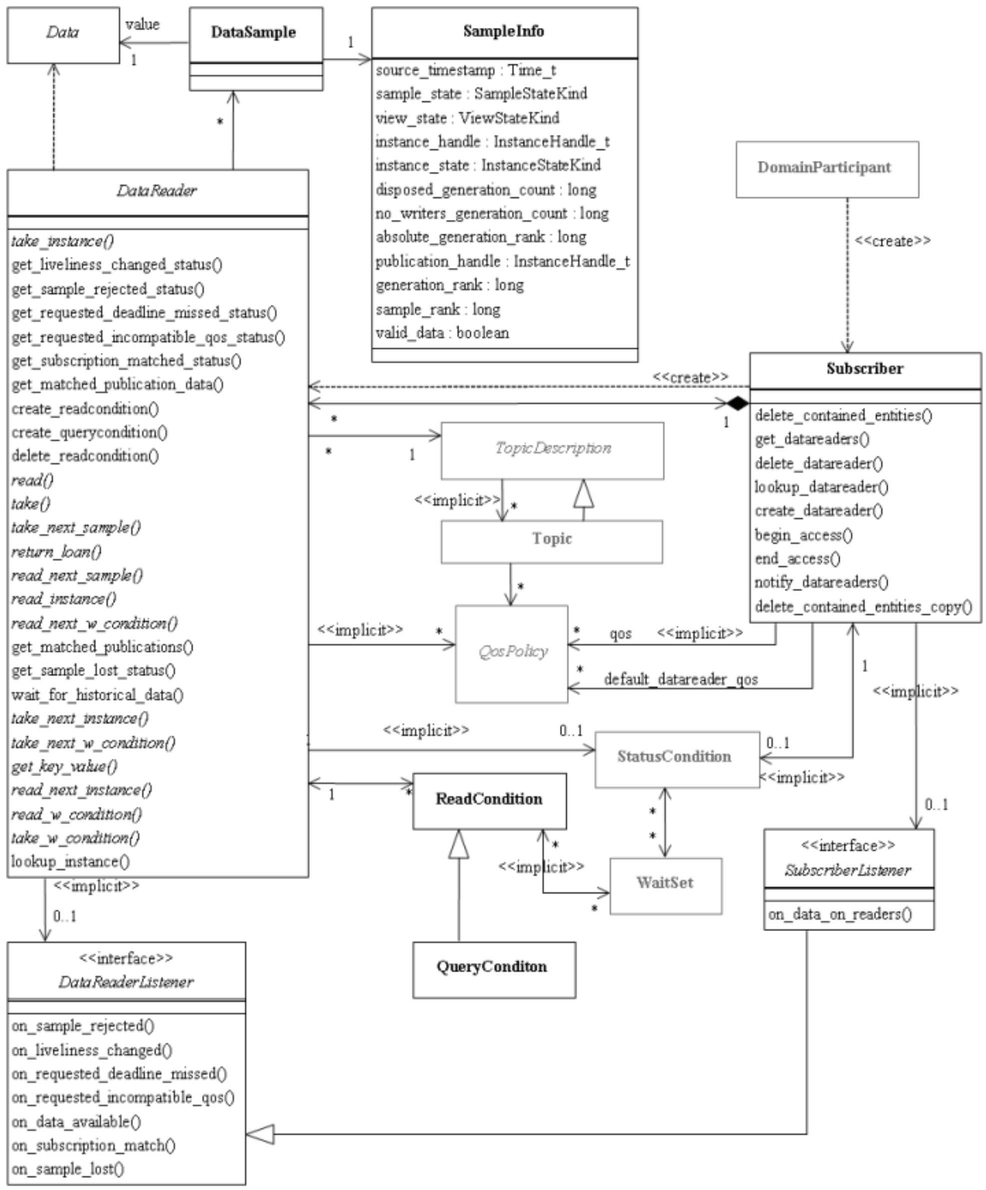


Table 7.1 **Subscriber Operations**

Working with ...	Operation	Description	Reference
DataReaders	begin_access	Indicates that the application is about to access the data samples in the *DataReaders* of the *Subscriber*.	*Currently not supported.*
	create_datareader	Creates a *DataReader*.	Section 7.3.1
	create_datareader_with_profile	Creates a *DataReader* with QoS from a specified QoS profile.	
	copy_from_topic_qos	Copies relevant QosPolicies from a *Topic* into a DataReaderQoS structure.	Section 7.2.4.5
	delete_contained_entities	Deletes all the *DataReaders* that were created by the *Subscriber*. Also deletes the corresponding *ReadConditions* created by the contained *DataReaders*.	Section 7.2.3.1
	delete_datareader	Deletes a specific *DataReader*.	Section 7.3.2
	end_access	Indicates that the application is done accessing the data samples in the *DataReaders* of the *Subscriber*.	*Currently not supported.*
	get_datareaders	Returns a list of *DataReaders* that contain samples with the specified sample_states, view_states and instance_states.	Section 7.2.6
	get_default_datareader_qos	Copies the *Subscriber's* default DataReaderQos values into a DataReaderQos structure.	Section 7.2.4
	get_status_changes	Gets all status changes.	Section 4.1.4
	lookup_datareader	Retrieves a *DataReader* previously created for a specific *Topic*.	Section 7.2.7
	notify_datareaders	Invokes the on_data_available() operation for attached *Listeners* of *DataReaders* that have new data samples.	Section 7.2.5
	set_default_datareader_qos	Sets or changes the *Subscriber's* default DataReaderQoS values.	Section 7.2.4

Table 7.1 **Subscriber Operations**

Working with ...	Operation	Description	Reference
Libraries and Profiles	get_default_library	Gets the *Subscriber's* default QoS profile library.	Section 7.2.4.3
	get_default_profile	Gets the *Subscriber's* default QoS profile.	
	get_default_profile_ library	Gets the library that contains the *Subscriber's* default QoS profile.	
	set_default_library	Sets the default library for a *Subscriber.*	
	set_default_profile	Sets the default profile for a *Subscriber.*	
Participants	get_participant	Gets the *Subscriber's DomainParticipant*.	Section 7.2.7
Subscribers	enable	Enables the *Subscriber.*	Section 4.1.2
	get_listener	Gets the currently installed *Listener.*	Section 7.2.5
	get_qos	Gets the *Subscriber's* current QosPolicy settings. This is most often used in preparation for calling set_qos.	Section 7.2.4.2
	set_listener	Sets the *Subscriber's Listener.* If you created the Subscriber without a *Listener,* you can use this operation to add one later.	Section 7.2.5
	set_qos	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.	Section 7.2.4.2
	set_qos_with_profile	Sets the *Subscriber's* QoS based on a QoS profile.	Section 7.2.4.2

7.2.1 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_SUBCRIBER_QOS_DEFAULT) and no Listener. The 'implicit *Subscriber*' can be accessed using the *DomainParticipant's* **get_implicit_subscriber()** operation (see Section 8.3.9).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.

7. Receiving Data

7.2.2 Creating Subscribers

Before you can explicitly create a *Subscriber*, you need a *DomainParticipant* (Section 8.3). To create a *Subscriber*, use the *DomainParticipant*'s **create_subscriber()** or **create_subscriber_with_profile()** operation:

```
DDSSubscriber* create_subscriber(
                                  const DDS_SubscriberQos &qos,
                                  DDSSubscriberListener *listener,
                                  DDS_StatusMask mask)

DDSSubscriber* create_subscriber_with_profile (
                                  const char *  library_name,
                                  const char *  profile_name,
                                  DDSSubscriberListener *  listener,
                                  DDS_StatusMask  mask )
```

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 Chapter 15: Configuring QoS with XML.

qos If you want the default QoS settings (described in the online documentation), use **DDS_SUBSCRIBER_QOS_DEFAULT** for this parameter (see Figure 7.2). If you want to customize any of the QosPolicies, supply a QoS structure (see Figure 7.3). The QoS structure for a *Subscriber* is described in Section 7.5.

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()**.

listener *Listeners* are callback routines. *RTI Data Distribution Service* uses them to notify your application when specific events (new 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 Section 7.2.5.

mask 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 Listeners (Section 4.4).

library_name A QoS Library is a named set of QoS profiles. See QoS Libraries (Section 15.9).

profile_name A QoS profile groups a set of related QoS, usually one per entity. See QoS Profiles (Section 15.8).

Figure 7.2 **Creating a Subscriber with Default QosPolicies**

```
// 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 Configuring QoS Settings when the Subscriber is Created (Section 7.2.4.1).

After you create a *Subscriber*, the next step is to use the *Subscriber* to create a *DataReader* for each *Topic*, see Section 7.3.1. For a list of operations you can perform with a *Subscriber*, see Table 7.1.

7.2.3 Deleting Subscribers

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 (Section 7.3.1) to delete them one at a time, or use the **delete_contained_entities()** operation (Section 7.2.3.1) to delete them all at the same time.

   ```
   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 Restricted Operations in Listener Callbacks (Section 4.5.1).

7.2.3.1 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*.

```
DDS_ReturnCode_t DDSSubscriber::delete_contained_entities ()
```

After this operation returns successfully, the application may delete the *Subscriber* (see Section 7.2.3).

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 samples.

7.2.4 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:

```
struct DDS_SubscriberQos {
    DDS_PresentationQosPolicy    presentation;
    DDS_PartitionQosPolicy       partition;
    DDS_GroupDataQosPolicy       group_data;
    DDS_EntityFactoryQosPolicy   entity_factory;
    DDS_ExclusiveAreaQosPolicy   exclusive_area;
};
```

Note: set_qos() cannot always be used by a *Listener*, see Restricted Operations in Listener Callbacks (Section 4.5.1).

Table 7.2 summarizes the meaning of each policy. *Subscribers* have the same set of QosPolicies as Publishers; they are described in detail in Publisher/Subscriber QosPolicies (Section 6.4). 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 online documentation for each policy.

Table 7.2 **Subscriber QosPolicies**

QosPolicy	Description
ENTITYFACTORY QosPolicy (Section 6.4.2)	Whether or not new entities created from this entity will start out as 'enabled.'
EXCLUSIVE_AREA QosPolicy (DDS Extension) (Section 6.4.3)	Whether or not the entity uses a multi-thread safe region with deadlock protection.
GROUP_DATA QosPolicy (Section 6.4.4)	A place to pass group-level information among applications. Usage is application-dependent.
PARTITION QosPolicy (Section 6.4.5)	Set of strings that introduces a logical partition among Topics visible by Publisher/Subscriber.
PRESENTATION QosPolicy (Section 6.4.6)	The order in which instance changes are presented to the Subscriber. By default, no order is used.

7.2.4.1 Configuring QoS Settings when the Subscriber is Created

As described in Creating Subscribers (Section 7.2.2), there are different ways to create a *Subscriber*, depending on how you want to specify its QoS (with or without a QoS Profile).

- ❑ In Figure 7.2 on page 7-11 we saw 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 Section 8.3.6.4).
- ❑ To create a *Subscriber* with non-default QoS settings, without using a QoS profile, see Figure 7.3 on page 7-14. 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 7.4 on page 7-14.
- ❑ 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 structure when calling **create_subscriber()**, as seen in Figure 7.5 on page 7-15.

For more information, see Creating Subscribers (Section 7.2.2) and Chapter 15: Configuring QoS with XML.

Figure 7.3 **Creating a Subscriber with Non-default QosPolicies (not from a profile)**

```
DDS_SubscriberQos subscriber_qos;1

// 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
}
```

1. For the C API, you need to use **DDS_SubscriberQos_INITIALIZER** or **DDS_SubscriberQos_initialize()**. See Special QosPolicy Handling Considerations for C (Section 4.2.2)

Figure 7.4 **Creating a Subscriber with a QoS Profile**

```
// create the subscriber with QoS profile
DDSSubscriber * subscriber =
           participant->create_subscriber_with_profile(
                                        "MySubscriberLibary",
                                        "MySubscriberProfile",
                                        NULL, DDS_STATUS_MASK_NONE);
if (subscriber == NULL) {
    // handle error
}
```

Figure 7.5 **Getting QoS Values from a Profile, Changing QoS Values, Creating a Subscriber with Modified QoS Values**

```
DDS_SubscriberQos subscriber_qos;1

// 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
}
```

1. For the C API, you need to use **DDS_SubscriberQos_INITIALIZER** or **DDS_SubscriberQos_initialize()**. See Special QosPolicy Handling Considerations for C (Section 4.2.2)

7.2.4.2 Changing QoS Settings After the 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 7.6 on page 7-16. 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.

7. Receiving Data

- ❑ 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 7.7 on page 7-16. For more information, see Chapter 15: Configuring QoS with XML.

Figure 7.6 **Changing the Qos of an Existing Subscriber**

```
DDS_SubscriberQos subscriber_qos;[1]
// 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
}
```

1. For the C API, you need to use **DDS_SubscriberQos_INITIALIZER** or **DDS_SubscriberQos_Initialize()**. See Special QosPolicy Handling Considerations for C (Section 4.2.2)

Figure 7.7 **Changing the QoS of an Existing Subscriber with a QoS Profile**

```
retcode = subscriber->set_qos_with_profile(
                   "SubscriberProfileLibrary","SubscriberProfile");

if (retcode != DDS_RETCODE_OK) {
    // handle error
}
```

7.2.4.3 Getting and Settings the 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 Chapter 15: Configuring QoS with XML.

```
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:

```
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_datareader_qos_with_profile()**, see Section 7.2.4.4 (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 Section 8.3.6.4.

7. Receiving Data

7.2.4.4 Getting and Setting Default QoS for DataReaders

These operations *set* the default QoS that will be used for new *DataReaders* if **create_datareader()** is called with DDS_DATAREADER_QOS_DEFAULT as the 'qos' parameter:

```
DDS_ReturnCode_t set_default_datareader_qos (
                    const DDS_DataReaderQos &qos)

DDS_ReturnCode_t set_default_datareader_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 *DataReaders* if **create_datareader()** is called with DDS_DATAREADER_QOS_DEFAULT as the 'qos' parameter:

```
DDS_ReturnCode_t get_default_datareader_qos (
                             DDS_DataReaderQos & qos)
```

The above operation gets the QoS settings that were specified on the last successful call to **set_default_datareader_qos()** or **set_default_datareader_qos_with_profile()**, or else, if the call was never made, the default values listed in DDS_DataReaderQos.

Note: It is not safe to set the default *DataReader* QoS values while another thread may be simultaneously calling **get_default_datareader_qos(), set_default_datareader_qos()** or **create_datareader()** 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_datareader_qos(),**

7.2.4.5 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 Section 5.1.3, 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 Section 7.3.7).

- **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.

7.2.5 Setting Up SubscriberListeners

Like all DDS Entities, Su*bscribers* 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 *RTI Data Distribution Service* 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 Section 4.4.

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

As illustrated in Figure 7.1, the S*ubscriberListener* interface extends the *DataReaderListener* interface. In other words, the *SubscriberListener* interface contains all the functions in the *DataReaderListener* interface. In addition, a *SubscriberListener* has 1 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 **DATA_AVAILABLE** status (Section 7.3.6.1) of DataReaders.

The *Subscriber*'s **DATA_ON_READERS** status is set whenever the **DATA_AVAILABLE** status is set for any of the *DataReaders* created by the *Subscriber*. This implies that one of its *DataReaders* has received new 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 **DATA_AVAILABLE** 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,

```
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
                      DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS;

subscriber =
    participant->create_subscriber(DDS_SUBSCRIBER_QOS_DEFAULT,
                                   listener, mask);
```

or

```
DDS_StatusMask mask = DDS_REQUESTED_DEADLINE_MISSED_STATUS |
                      DDS_REQUESTED_INCOMPATIBLE_QOS_STATUS;

subscriber->set_listener(listener, mask);
```

As previously mentioned, the callbacks in the *SubscriberListener* act as 'default' callbacks for all the *DataReader*s contained within. When *RTI Data Distribution Service* 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, *RTI Data Distribution Service* dispatches the event to the *DataReaderListener* callback. Otherwise, *RTI Data Distribution Service* 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*, *RTI Data Distribution Service* first checks to see if the *SubscriberListener* has **DATA_ON_READERS** enabled. If so, *RTI Data Distribution Service* will invoke the **on_data_on_readers()** callback. Otherwise, *RTI Data Distribution Service* 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 Section 7.3.3.

Similarly, the callbacks in the *DomainParticipantListener* act as 'default' callbacks for all the *Subscribers* that belong to it. For more information on *DomainParticipantListeners*, see Section 8.3.5.

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 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 7.8 shows a *SubscriberListener* that simply notifies its *DataReaders* when new data arrives.

Figure 7.8 **Simple SubscriberListener**

```
class MySubscriberListener : public DDSSubscriberListener {
  public:
    void on_data_on_readers(DDSSubscriber *);

    /* For this example we take no action other operations */
};

void MySubscriberListener::on_data_on_readers
                                 (DDSSubscriber *subscriber)
{
    ... // do global processing

    // now dispatch data arrival event to specific DataReaders
    subscriber->notify_datareaders();
}
```

7.2.6 Getting DataReaders with Specific Samples

The *Subscriber's* **get_datareaders()** operation retrieves a list of *DataReaders* that have samples with specific **sample_states**, **view_states**, and i**nstance_states**.

```
DDS_ReturnCode_t get_datareaders (DDSDataReaderSeq & readers,
                            DDS_SampleStateMask    sample_states,
                            DDS_ViewStateMask     view_states,
                            DDS_InstanceStateMask  instance_states)
```

Qualified *DataReaders* (those with samples that match the desired states) will appear in the returned list, in an unspecified order.

For more information, see The SampleInfo Structure (Section 7.4.5).

7.2.7 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 builtin *DataReaders* for the builtin topics. The built-in *DataReader* is created when this operation is called on a builtin topic for the first time.

  If you are going to modify the transport properties for the builtin *DataReaders*, do so *before* using this operation. Builtin transports are implicitly registered when the *DomainParticipant* is enabled or the first *DataWriter/DataReader* is created. To ensure that builtin *DataReaders* receive all the discovery traffic, you should lookup the *DataReader* before the *DomainParticipant* is enabled. Therefore the suggested sequence when looking up builtin *DataReaders* is:

  1. Create a disabled *DomainParticipant* (see Section 6.4.2).
  2. If you want to use non-default values, modify the builtin transport properties (see Section 13.5).
  3. Call **get_builtin_subscriber()** (see Section 14.2).
  4. Call **lookup_datareader()**.
  5. Call **enable()** on the *DomainParticipant* (see Section 4.1.2).
- **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.

7. Receiving Data

7.2.8 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** (Section 7.2.8.1). The following statuses can be monitored by the *SubscriberListener*.

- ❑ DATA_ON_READERS Status (Section 7.2.8.1)
- ❑ DATA_AVAILABLE Status (Section 7.3.6.1)
- ❑ LIVELINESS_CHANGED Status (Section 7.3.6.4)
- ❑ REQUESTED_DEADLINE_MISSED Status (Section 7.3.6.5)
- ❑ REQUESTED_INCOMPATIBLE_QOS Status (Section 7.3.6.6)
- ❑ SAMPLE_LOST Status (Section 7.3.6.7)
- ❑ SAMPLE_REJECTED Status (Section 7.3.6.8)
- ❑ SUBSCRIPTION_MATCHED Status (Section 7.3.6.9)

You can access *Subscriber* status by using a *SubscriberListener* or its inherited **get_status_changes()** operation (see Section 4.1.4), which can be used to explicitly poll for the **DATA_ON_READERS** status of the *Subscriber*.

7.2.8.1 DATA_ON_READERS Status

The **DATA_ON_READERS** status, like the **DATA_AVAILABLE** status for *DataReaders*, is a *read* communication status, which makes it somewhat different from other *plain* communication statuses. (See Types of Communication Status (Section 4.3.1) 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 **DATA_AVAILABLE** 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 the 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.

7.3 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 data samples you want to receive.

After you create a *DataReader*, you will be able to use the operations listed in Table 7.3. 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 data samples. For more details on all operations, see the online documentation. The *DataReaderListener* is described in Section 7.3.3.

DataReaders are created by using operations on a *DomainParticipant* or a *Subscriber*, as described in Section 7.2.1. If you use the *DomainParticipant's* operations, the *DataReader* will belong to an implicit *Subscriber* that is automatically created by the middleware. If you use a *Subscriber's* operations, the *DataReader* will belong to that *Subscriber*. So either way, the *DataReader* belongs to a *Subscriber*.

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

Table 7.3 **DataReader Operations**

Purpose	Operation	Description	Reference
Configuring the DataReader	enable	Enables the *DataReader*.	Section 4.1.2
	get_qos	Gets the QoS.	Section 7.3.7
	set_qos	Modifies the QoS.	
	set_qos_with_profile	Modifies the QoS based on a QoS profile.	
	get_listener	Gets the currently installed *Listener*.	Section 7.3.3
	set_listener	Replaces the *Listener*.	

Table 7.3 **DataReader Operations**

Purpose	Operation	Description	Reference
Accessing Data Samples with "Read" *(Use FooDataReader, see Section 7.4.3)*	read	Reads (copies) a collection of data samples from the *DataReader.*	Section 7.4.3
	read_instance	Identical to read, but all samples returned belong to a single instance, which you specify as a parameter.	Section
	read_next_instance	Similar to read_instance, but the actual instance is not directly specified as a parameter. Instead, the samples will all belong to instance ordered after the one previously read.	Section 7.4.3.5
	read_next_instance_w_condition	Accesses a collection of data samples of the next instance that match a specific set of *ReadConditions*, from the *DataReader.*	Section 7.4.3.7
	read_next_sample	Reads the next not-previously-accessed data value from the *DataReader.*	Section 7.4.3.3
	read_w_condition	Accesses a collection of data samples from the *DataReader* that match specific *ReadCondition* criteria.	Section 7.4.3.6
Accessing Data Samples with "Take" *(Use FooDataReader, see Section 7.4.3)*	take	Like read, but the samples are removed from the *DataReader's* receive queue.	Section 7.4.3
	take_instance	Like read_instance, but the samples are removed from the *DataReader's* receive queue.	Section 7.4.3.4
	take_next_instance	Like read_next_instance, but the samples are removed from the *DataReader's* receive queue.	Section 7.4.3.5
	take_next_instance_w_condition	Accesses (and removes) a collection of data samples of the next instance that match a specific set of *ReadConditions*, from the *DataReader.*	Section 7.4.3.7
	take_next_sample	Like read_next_sample, but the samples are removed from the *DataReader's* receive queue.	Section 7.4.3.3
	take_w_condition	Accesses (and removes) a collection of data samples from the *DataReader* that match specific *ReadCondition* criteria.	Section 7.4.3.6

Table 7.3 **DataReader Operations**

Purpose	Operation	Description	Reference
Working with Data Samples and FooDataReader *(Use FooDataReader, see Section 7.4.3)*	narrow	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.	Section 7.4.1
	return_loan	Returns buffers loaned in a previous read or take call.	Section 7.4.2
	get_key_value	Gets the key for an instance handle.	Section 7.3.8.4
	lookup_instance	Gets the instance handle that corresponds to an instance key.	Section 7.3.8.3
Checking Status	get_liveliness_changed_status	Gets LIVELINESS_CHANGED_STATUS status.	Section 7.3.6
	get_requested_deadline_missed_status	Gets REQUESTED_DEADLINE_MISSED_STATUS status.	
	get_requested_incompatible_qos_status	Gets REQUESTED_INCOMPATIBLE_QOS_STATUS status.	
	get_sample_lost_status	Gets SAMPLE_LOST_STATUS status.	
	get_sample_rejected_status	Gets SAMPLE_REJECTED_STATUS status.	
	get_subscription_matched_status	Gets SUBSCRIPTION_MATCHED_STATUS status.	
	get_status_changes	Gets a list of statuses that changed since last time the application read the status or the listeners were called.	Section 4.1.4
	get_datareader_cache_status	Gets DATA_READER_CACHE_STATUS status.	Section 7.3.4 Section 7.3.6
	get_datareader_protocol_status	Gets DATA_READER_PROTOCOL_STATUS status.	
	get_matched_publication_datareader_protocol_status	Get the protocol status for this *DataReader*, per matched publication identified by the publication_handle.	

Table 7.3 **DataReader Operations**

Purpose	Operation	Description	Reference
Navigating Relationships	get_instance_handle	Returns the DDS_InstanceHandle_t associated with the Entity.	Section 4.1.3
	get_matched_publication_data	Gets information on a publication with a matching Topic and compatible QoS.	Section 7.3.8.1
	get_matched_publications	Gets a list of publications that have a matching Topic and compatible QoS. These are the publications currently associated with the *DataReader*.	
	get_subscriber	Gets the *Subscriber* that created the *DataReader*.	Section 7.3.8.2
	get_topicdescription	Gets the Topic associated with the *DataReader*.	
Working with Conditions	create_querycondition	Creates a *QueryCondition*.	Section 4.6.7
	create_readcondition	Creates a *ReadCondition*.	
	delete_readcondition	Deletes a *ReadCondition*/*QueryCondition* attached to the *DataReader*.	
	delete_contained_entities	Deletes all the *ReadConditions/QueryConditions* that were created by means of the "create" operations on the *DataReader*.	Section 7.3.2.1
	get_statuscondition	Gets the StatusCondition associated with the Entity.	Section 4.6.8
Waiting for Historical Data	wait_for_historical_data	Waits until all "historical" (previously sent) data is received. Only valid for Reliable *DataReaders* with non-VOLATILE DURABILITY.	Section 7.3.5

7.3.1 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 Section 7.2.1. If you use a *Subscriber's* operations to create a *DataReader*, it will belong to that *Subscriber*.

```
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)
```

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 Chapter 15: Configuring QoS with XML.

topic The *Topic* to which the *DataReader* is subscribing. This must have been previously created by the same *DomainParticipant*.

qos If you want the default QoS settings (described in the online documentation), use **DDS_DATAREADER_QOS_DEFAULT** for this parameter (see Figure 7.9 on page 7-30). If you want to customize any of the QosPolicies, supply a QoS structure (see Section 7.3.7).

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 data samples arrive. *RTI Data Distribution Service* 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 Section 7.3.3 and Section 7.3.6.

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 Section 7.3.3 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 Listeners (Section 4.4).

library_name A QoS Library is a named set of QoS profiles. See QoS Libraries (Section 15.9).

profile_name A QoS profile groups a set of related QoS, usually one per entity. See QoS Profiles (Section 15.8).

After you create a *DataReader*, you can use it to retrieve received data. See Section 7.4.

Note: When a *DataReader* is created, only those transports already registered are available to the *DataReader*. The builtin transports are implicitly registered when (a) the *DomainParticipant* is enabled, (b) the first *DataReader* is created, or (c) you lookup a builtin *DataReader*, whichever happens first.

Figure 7.9 shows an example of how to create a *DataReader* with default QosPolicies.

Figure 7.9 **Creating a DataReader with Default QosPolicies**

```
// 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 *DataWriter*, see Configuring QoS Settings when the DataReader is Created (Section 7.3.7.1)

7.3.2 Deleting DataReaders

To delete a *DataReader*:

1. 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 (Section 7.3.2.1) to delete them all at the same time.

```
DDS_ReturnCode_t delete_readcondition (DDSReadCondition *condition)
```

2. Delete the *DataReader* by using the *Subscriber's* **delete_datareader()** operation (Section 7.2.3).

Note: A *DataReader* cannot be deleted within its own reader listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

To delete all of a *Subscriber's DataReaders,* use the *Subscriber's* **delete_contained_entities()** operation (see Section 7.2.3.1).

7.3.2.1 Deleting Contained ReadConditions

The *DataReader's* **delete_contained_entities()** operation deletes all the *ReadConditions* and *QueryConditions* (Section 4.6.7) that were created by the *DataReader.*

```
DDS_ReturnCode_t delete_contained_entities ()
```

After this operation returns successfully, the application may delete the *DataReader* (see Section 7.3.2).

7.3.3 Setting Up DataReaderListeners

DataReaders may optionally have *Listeners.* A *DataReaderListener* is a collection of callback methods; these methods are invoked by *RTI Data Distribution Service* when data samples are received or when there are status changes for the *DataReader.*

Note: Some operations cannot be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

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 Section 7.2.5 and Section 8.3.5).

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 Section 7.4 to access the data that has been received.

Listeners are typically set up when the *DataReader* is created (see Section 7.3.1). You can also set one up after creation by using the *DataReader*'s **get_listener()** and **set_listener()** operations. *RTI Data Distribution Service* will invoke a *DataReader*'s *Listener* to report the status changes listed in Table 7.4 (if the Listener is set up to handle the particular status, see Section 7.3.3).

Table 7.4 **DataReaderListener Callbacks**

This DataReaderListener callback...	...is triggered by a change in this status:
on_data_available()	DATA_AVAILABLE Status (Section 7.3.6.1)
on_liveliness_changed()	LIVELINESS_CHANGED Status (Section 7.3.6.4)
on_requested_deadline_missed()	REQUESTED_DEADLINE_MISSED Status (Section 7.3.6.5)
on_requested_incompatible_qos()	REQUESTED_INCOMPATIBLE_QOS Status (Section 7.3.6.6)
on_sample_lost()	SAMPLE_LOST Status (Section 7.3.6.7)
on_sample_rejected()	SAMPLE_REJECTED Status (Section 7.3.6.8)
on_subscription_matched()	SUBSCRIPTION_MATCHED Status (Section 7.3.6.9)

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 Listeners (Section 4.4).

Figure 7.10 shows a *DataReaderListener* that simply prints the data it receives.

Figure 7.10 **Simple DataReaderListener**

```
class MyReaderListener : public DDSDataReaderListener {
  public:
    virtual void on_data_available(DDSDataReader* reader);

    // don't do anything for the other callbacks
};
void MyReaderListener::on_data_available(DDSDataReader* reader)
{
    FooDataReader *Foo_reader = NULL;
    FooSeq data_seq;              // In C, sequences have to be initialized
    DDS_SampleInfoSeq info_seq;   // before use, see Section 7.4.4.
    DDS_ReturnCode_t retcode;
    int i;
    // Must cast generic reader into reader of specific type
    Foo_reader = FooDataReader::narrow(reader);
    if (Foo_reader == NULL) {
        printf("DataReader narrow error\n");
        return;
    }
    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) {
        printf("take error %d\n", retcode);
        return;
    }
    for (i = 0; i < data_seq.length(); ++i) {
        // the data may not be valid if the sample is meta information
        // about the creation or deletion of an instance
        if (info_seq[i].valid_data) {
            FooTypeSupport::print_data(&data_seq[i]);
        }
    }
    // RTI Data Distribution Service gave a pointer to internal memory via
    // take(), must return the memory when finished processing the data
    retcode = Foo_reader->return_loan(data_seq, info_seq);
    if (retcode != DDS_RETCODE_OK) {
        printf("return loan error %d\n", retcode);
    }
}
```

7.3.4 Checking DataReader Status and StatusConditions

You can access individual communication status for a *DataReader* with the operations shown in Table 7.5.

Table 7.5 **DataReader Status Operations**

Use this operation...	...to retrieve this status:
get_datareader_cache_status	DATA_READER_CACHE_STATUS (Section 7.3.6.2)
get_datareader_protocol_status	DATA_READER_PROTOCOL_STATUS (Section 7.3.6.3)
get_matched_publication_ datareader_protocol_status	
get_liveliness_changed_status	LIVELINESS_CHANGED Status (Section 7.3.6.4)
get_sample_lost_status	SAMPLE_LOST Status (Section 7.3.6.7)
get_sample_rejected_status	SAMPLE_REJECTED Status (Section 7.3.6.8)
get_requested_deadline_missed_status	REQUESTED_DEADLINE_MISSED Status (Section 7.3.6.5)
get_requested_incompatible_qos_status	REQUESTED_INCOMPATIBLE_QOS Status (Section 7.3.6.6)
get_subscription_match_status	SUBSCRIPTION_MATCHED Status (Section 7.3.6.9)
get_status_changes	All of the above
get_statuscondition	See StatusConditions (Section 4.6.8)

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 *RTI Data Distribution Service* returns from calling the *Listener* callback associated with that status.

For more on status, see Setting Up DataReaderListeners (Section 7.3.3), Statuses for DataReaders (Section 7.3.6), and Listeners (Section 4.4).

7.3.5 Waiting for Historical Data

The **wait_for_historical_data()** operation waits (blocks) until all "historical" data is received. "Historical" data means samples that were written before the *DataReader* joined the domain.

This operation is intended only for *DataReaders* that have:

- ❑ DURABILITY QosPolicy (Section 6.5.6) **kind** set to TRANSIENT_LOCAL (not VOLATILE)
- ❑ RELIABILITY QosPolicy (Section 6.5.17) **kind** set to RELIABLE.

  Calling **wait_for_historical_data()** on a non-reliable *DataReader* will always return immediately, since *RTI Data Distribution Service* will never deliver historical data to non-reliable *DataReaders*.

As soon as an application enables a non-VOLATILE *DataReader*, it will start receiving both "historical" data as well as any new data written by matching *DataWriters*. If you want the subscribing application to wait until all "historical" data is received, use this operation:

```
DDS_ReturnCode_t wait_for_historical_data
                                   (const DDS_Duration_t &  max_wait)
```

The **wait_for_historical_data()** operation blocks the calling thread until either all "historical" data is received, or else duration specified by the **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 **max_wait** elapsed before all the data was received.

7.3.6 Statuses for DataReaders

There are several types of statuses available for a *DataReader*. You can use the **get_*_status()** operations (Section 7.3.4) to access them, use a *DataReaderListener* (Section 7.3.3) to listen for changes in their values (for those statuses that have Listeners), or use a *StatusCondition* and a *WaitSet* (Section 4.6.8) to wait for changes. Each status has an associated data structure and is described in more detail in the following sections.

- ❑ DATA_AVAILABLE Status (Section 7.3.6.1)
- ❑ DATA_READER_CACHE_STATUS (Section 7.3.6.2)
- ❑ DATA_READER_PROTOCOL_STATUS (Section 7.3.6.3)

- ❏ LIVELINESS_CHANGED Status (Section 7.3.6.4)
- ❏ REQUESTED_DEADLINE_MISSED Status (Section 7.3.6.5)
- ❏ REQUESTED_INCOMPATIBLE_QOS Status (Section 7.3.6.6)
- ❏ SAMPLE_LOST Status (Section 7.3.6.7)
- ❏ SAMPLE_REJECTED Status (Section 7.3.6.8)
- ❏ SUBSCRIPTION_MATCHED Status (Section 7.3.6.9)

7.3.6.1 DATA_AVAILABLE Status

This status indicates that new data is available for the *DataReader*. In most cases, this means that one new sample has been received. However, there are situations in which more than one samples for the *DataReader* may be received before the **DATA_AVAILABLE** status changes. For example, if the *DataReader* has the DURABILITY QosPolicy (Section 6.5.6) set to be non-**VOLATILE**, then the *DataReader* may receive a batch of old data samples all at once. Or if data is being received reliably from *DataWriters*, *RTI Data Distribution Service* may present several 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 Section 7.2.8 and Section 4.3.1 for more information on read communication statuses.

The *DataReaderListener's* **on_data_available()** callback is invoked when this status changes, unless the *SubscriberListener* (Section 7.2.5) or *DomainParticipantListener* (Section 8.3.5) has implemented an **on_data_on_readers()** callback. In that case, **on_data_on_readers()** will be invoked instead.

7.3.6.2 DATA_READER_CACHE_STATUS

This status keeps track of the number of samples in the reader's cache.

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 7.6.

Table 7.6 **DDS_DataReaderCacheStatus**

Type	Field Name	Description
DDS_Long	sample_count_peak	Highest number of samples in the *DataReader's* queue over the lifetime of the *DataReader.*
DDS_Long	sample_count	Current number of samples in the *DataReader's* queue. Includes samples that may not yet be available to be read or taken by the user due to samples being received out of order or settings in the PRESENTATION QosPolicy (Section 6.4.6).

7.3.6.3 DATA_READER_PROTOCOL_STATUS

The status of a *DataReader's* internal protocol related metrics (such as the number of samples received, filtered, rejected) and the status of wire protocol traffic. The structure for this status appears in Table 7.7 on page 7-38.

This status does not have an associated Listener. You can access this status by calling the following operations on the *DataReader* (all of which return the status structure described in Table 7.7):

- ❏ **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**.

Note: Status for a remote entity is only kept while the entity is alive. Once a remote entity is no longer alive, its status is deleted. If you try to get the matched subscription status for a remote entity that is no longer alive, the 'get status' call will return an error.

7.3.6.4 LIVELINESS_CHANGED Status

This status indicates that the liveliness of one or more matched *DataWriters* has changed (i.e., one or more *DataWriters* has become alive or not alive). The mechanics of determining liveliness between a *DataWriter* and a *DataReader* is specified in their LIVELINESS QosPolicy (Section 6.5.11).

The structure for this status appears in Table 7.8.

The *DataReaderListener's* **on_liveliness_changed()** callback is invoked when this status changes. You can also retrieve the value by calling the *DataReader's* **get_liveliness_changed_status()** operation.

This status is reciprocal to the RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.5.8) for a *DataWriter.*

Table 7.7 **DDS_DataReaderProtocolStatus**

Type	Field Name	Description
DDS_LongLong	received_sample_count	The number of user samples from a remote *DataWriter* received for the first time by a local *DataReader*.
	received_sample_count_change	The incremental change in the number of user samples from a remote *DataWriter* received for the first time by a local *DataReader* since the last time the status was read.
	received_sample_bytes	The number of bytes of user samples from a remote *DataWriter* received for the first time by a local *DataReader*.
	received_sample_bytes_change	The incremental change in the number of bytes of user samples from a remote *DataWriter* received for the first time by a local *DataReader* since the last time the status was read.
DDS_LongLong	duplicate_sample_count	The number of samples from a remote *DataWriter* received, not for the first time, by a local *DataReader*.
	duplicate_sample_count_change	The incremental change in the number of samples from a remote *DataWriter* received, not for the first time, by a local *DataReader* since the last time the status was read.
	duplicate_sample_bytes	The number of bytes of samples from a remote *DataWriter* received, not for the first time, by a local *DataReader*.
	duplicate_sample_bytes_change	The incremental change in the number of bytes of samples from a remote *DataWriter* received, not for the first time, by a local *DataReader* since the last time the status was read.

Table 7.7 **DDS_DataReaderProtocolStatus**

Type	Field Name	Description
DDS_LongLong	filtered_sample_count	The number of user samples filtered by the local *DataReader* due to ContentFilteredTopics or Time-Based Filter.
	filtered_sample_count_ change	The incremental change in the number of user samples filtered by the local *DataReader* due to Content-FilteredTopics or Time-Based Filter since the last time the status was read.
	filtered_sample_bytes	The number of bytes of user samples filtered by the local DataReader due to ContentFilteredTopics or Time-Based Filter.
	filtered_sample_bytes_ change	The incremental change in the number of bytes of user samples filtered by the local *DataReader* due to ContentFilteredTopics or Time-Based Filter since the last time the status was read.
DDS_LongLong	received_heartbeat_count	The number of Heartbeats from a remote *DataWriter* received by a local *DataReader*.
	received_heartbeat_count_ change	The incremental change in the number of Heartbeats from a remote *DataWriter* received by a local *DataReader* since the last time the status was read.
	received_heartbeat_bytes	The number of bytes of Heartbeats from a remote *DataWriter* received by a local DataReader.
	received_heartbeat_bytes_ change	The incremental change in the number of bytes of Heartbeats from a remote *DataWriter* received by a local *DataReader* since the last time the status was read.

Table 7.7 **DDS_DataReaderProtocolStatus**

Type	Field Name	Description
DDS_LongLong	sent_ack_count	The number of ACKs sent from a local *DataReader* to a matching remote *DataWriter*.
	sent_ack_count_change	The incremental change in the number of ACKs sent from a local *DataReader* to a matching remote *DataWriter* since the last time the status was read.
	sent_ack_bytes	The number of bytes of ACKs sent from a local *DataReader* to a matching remote *DataWriter*.
	sent_ack_bytes_change	The incremental change in the number of bytes of ACKs sent from a local *DataReader* to a matching remote *DataWriter* since the last time the status was read.
DDS_LongLong	sent_nack_count	The number of NACKs sent from a local *DataReader* to a matching remote *DataWriter*.
	sent_nack_count_change	The incremental change in the number of NACKs sent from a local *DataReader* to a matching remote *DataWriter* since the last time the status was read.
	sent_nack_bytes	The number of bytes of NACKs sent from a local *DataReader* to a matching remote *DataWriter*.
	sent_nack_bytes_change	The incremental change in the number of bytes of NACKs sent from a local *DataReader* to a matching remote *DataWriter* since the last time the status was read.

Table 7.7 **DDS_DataReaderProtocolStatus**

Type	Field Name	Description
DDS_LongLong	received_gap_count	The number of GAPs received from remote *DataWriter* to this *DataReader*.
	received_gap_count_change	The incremental change in the number of GAPs received from remote *DataWriter* to this *DataReader* since the last time the status was read.
	received_gap_bytes	The number of bytes of GAPs received from remote *DataWriter* to this *DataReader*.
	received_gap_bytes_change	The incremental change in the number of bytes of GAPs received from remote *DataWriter* to this *DataReader* since the last time the status was read.
DDS_LongLong	rejected_sample_count	The number of times a sample is rejected for unanticipated reasons in the receive path.
	rejected_sample_ count_change	The incremental change in the number of times a sample is rejected for unanticipated reasons in the receive path since the last time the status was read.
DDS_ SequenceNumber_t	first_available_sample_ sequence_number	Sequence number of the first available sample in a matched *DataWriter's* reliability queue. Applicable only when retrieving matched *DataWriter* statuses.
	last_available_sample_ sequence_number	Sequence number of the last available sample in a matched *DataWriter's* reliability queue. Applicable only when retrieving matched *DataWriter* statuses.
	last_committed_sample_ sequence_number	Sequence number of the last committed sample (i.e. available to be read or taken) in a matched *DataWriter's* reliability queue. Applicable only when retrieving matched *DataWriter* statuses.

Table 7.7 **DDS_DataReaderProtocolStatus**

Type	Field Name	Description
DDS_Long	uncommitted_sample_count	Number of received 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.

Table 7.8 **DDS_LivelinessChangedStatus**

Type	Field Name	Description
DDS_Long	alive_count	Number of matched *DataWriters* that are currently alive.
	not_alive_count	Number of matched *DataWriters* that are not currently alive.
	alive_count_change	The change in the alive_count since the last time the *Listener* was called or the status was read.
	not_alive_count_change	The change in the not_alive_count since the last time the *Listener* was called or the status was read.
DDS_InstanceHandle_t	last_publication_handle	A handle to the last *DataWriter* to change its liveliness.

7.3.6.5 REQUESTED_DEADLINE_MISSED Status

This status indicates that the *DataReader* did not receive a new sample for an data-instance within the time period set in the *DataReader's* DEADLINE QosPolicy (Section 6.5.4). 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 sample within the DEADLINE period. For more information about keys and instances, see Section 2.2.2.

The structure for this status appears in Table 7.9.

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.

Table 7.9 **DDS_RequestedDeadlineMissedStatus**

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times that the deadline was violated for any instance read by the *DataReader*.
DDS_Long	total_count_change	The change in total_count since the last time the *Listener* was called or the status was read.
DDS_Instance Handle_t	last_instance_handle	Handle to the last data-instance in the *DataReader* for which a requested deadline was missed.

7.3.6.6 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 7.10.

Table 7.10 **DDS_RequestedIncompatibleQosStatus**

Type	Field Name	Description
DDS_Long	total_count	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*.
DDS_Long	total_count_change	The change in total_count since the last time the *Listener* was called or the status was read.
DDS_QosPolicyId_t	last_policy_id	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.)
DDS_QosPolicyCountSeq	policies	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*.

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.

7.3.6.7 SAMPLE_LOST Status

This status indicates that one or more samples written by a matched *DataWriter* have failed to be received.

For a *DataReader*, when there are insufficient resources to accept incoming samples of data, samples may be dropped by the receiving application. Those samples are considered to be REJECTED (see Section 7.3.6.8). But *DataWriters* are limited in the number of published data samples that they can store, so that if a *DataWriter* continues to publish data samples, new data may overwrite old data that have not yet been received by the *DataReader*. The samples that are overwritten can never be resent to the *DataReader* and thus are considered to be *lost*.

This status applies to reliable *and* best-effort *DataReaders*, see the RELIABILITY QosPolicy (Section 6.5.17).

The structure for this status appears in Table 7.9.

Table 7.11 **DDS_SampleLostStatus**

Type	Field Name	Description
DDS_Long	total_count	Cumulative count of all the samples that have been lost, across all instances of data written for the *Topic*.
	total_count_change	The incremental number of samples lost since the last time the *Listener* was called or the status was read.

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.

7.3.6.8 SAMPLE_REJECTED Status

This status indicates that one or more samples received from a matched *DataWriter* have been dropped by the *DataReader* because a resource limit would have been exceeded. For example, if the receive queue is full, the number of samples in the queue is equal to the **max_samples** parameter of the RESOURCE_LIMITS QosPolicy (Section 6.5.18).

The structure for this status appears in Table 7.12. The reason the sample was rejected appears in the **last_reason** field. The possible values are listed in Table 7.13.

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.

Table 7.12 **DDS_SampleRejectedStatus**

Type	Field Name	Description
DDS_Long	total_count	Cumulative count of all the samples that have been rejected by the *DataReader*.
	total_count_change	The incremental number of samples rejected since the last time the *Listener* was called or the status was read.
	current_count	The current number of writers with which the *DataReader* is matched.
	current_count_change	The change in current_count since the last time the *Listener* was called or the status was read.
DDS_SampleRejectedStatus Kind	last_reason	Reason for rejecting the last sample. See Table 7.13.
DDS_InstanceHandle_t	last_instance_handle	Handle to the data-instance for which the last sample was rejected.

Table 7.13 **DDS_SampleRejectedStatusKind**

Reason Kind	Description	Related QosPolicy
DDS_NOT_REJECTED	Sample was accepted.	
DDS_REJECTED_BY_ INSTANCE_LIMIT	A resource limit on the number of instances that can be handled at the same time by the *DataReader* was reached.	RESOURCE_LIMITS QosPolicy (Section 6.5.18)
DDS_REJECTED_BY_ REMOTE_WRITERS_LIMIT	A resource limit on the number of *DataWriters* from which a *DataReader* may read was reached.	DATA_READER_RESOURCE_ LIMITS QosPolicy (DDS Extension) (Section 7.6.2)
DDS_REJECTED_BY_ REMOTE_WRITERS_PER_ INSTANCE_LIMIT	A resource limit on the number of *DataWriters* for a single instance from which a DataReader may read was reached.	DATA_READER_RESOURCE_ LIMITS QosPolicy (DDS Extension) (Section 7.6.2)
DDS_REJECTED_BY_ SAMPLES_LIMIT	A resource limit on the total number of samples was reached.	RESOURCE_LIMITS QosPolicy (Section 6.5.18)

Table 7.13 **DDS_SampleRejectedStatusKind**

Reason Kind	Description	Related QosPolicy
DDS_REJECTED_BY_ SAMPLES_PER_ INSTANCE_LIMIT	A resource limit on the number of samples per instance was reached.	RESOURCE_LIMITS QosPolicy (Section 6.5.18)
DDS_REJECTED_BY_ SAMPLES_PER_ REMOTE_WRITER_LIMIT	A resource limit on the number of samples that a *DataReader* may store from a specific *DataWriter* was reached.	DATA_READER_RESOURCE_ LIMITS QosPolicy (DDS Extension) (Section 7.6.2)

7.3.6.9 SUBSCRIPTION_MATCHED 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 data type (implied by having the same *Topic*), and compatible QosPolicies. In addition, if user code has directed *RTI Data Distribution Service* to ignore certain *DataWriters*, then those *DataWriters* will never be matched. See Section 14.4.2 for more on setting up a *DomainParticipant* to ignore specific *DataWriters*.

The structure for this status appears in Table 7.14.

Table 7.14 **DDS_SubscriptionMatchedStatus**

Type	Field Name	Description
DDS_Long	total_count	Cumulative number of times the *DataReader* discovered a "match" with a *DataWriter*.
	total_count_change	The change in total_count since the last time the *Listener* was called or the status was read.
	current_count	The number of *DataWriters* currently matched to the concerned *DataReader*.
	current_count_change	The change in current_count since the last time the listener was called or the status was read.
	current_count_peak	The highest value that current_count has reached until now.
DDS_Instance Handle_t	last_publication_ handle	Handle to the last *DataWriter* that matched the *DataReader* causing the status to change.

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.

7.3.7 Setting DataReader QosPolicies

A *DataReader*'s QosPolicies control its behavior. Think of QosPolicies as the 'properties' for the *DataReader*. The **DDS_DataReaderQos** structure has the following format:

```
struct DDS_DataReaderQos {
    DDS_DurabilityQosPolicy          durability;
    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_TimeBasedFilterQosPolicy     time_based_filter;
    DDS_ReaderDataLifecycleQosPolicy reader_data_lifecycle;
    // Extensions to the DDS standard:
    DDS_DataReaderResourceLimitsQosPolicy reader_resource_limits;
    DDS_DataReaderProtocolQosPolicy  protocol;
    DDS_TransportSelectionQosPolicy  transport_selection;
    DDS_TransportUnicastQosPolicy    unicast;
    DDS_TransportMulticastQosPolicy  multicast;
    DDS_TypeSupportQosPolicy          type_support;
    DDS_PropertyQosPolicy             property;
};
```

Note: set_qos() cannot always be used within a listener callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

Table 7.15 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 online documentation.

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 QoS Requested vs. Offered Compatibility—the RxO Property (Section 4.2.1).

Table 7.15 **DataReader QosPolicies**

QosPolicy	Description
DataReaderProtocol	This QosPolicy configures the DDS on-the-network protocol, RTPS. See Section 7.6.1.
DataReaderResourceLimits	Various settings that configure how *DataReaders* allocate and use physical memory for internal resources. See Section 7.6.2.
Deadline	For a *DataReader*, specifies the maximum expected elapsed time between arriving data samples. For a *DataWriter*, specifies a commitment to publish samples with no greater elapsed time between them. See Section 6.5.4.
DestinationOrder	Controls how *RTI Data Distribution Service* 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 Section 6.5.5.
Durability	Specifies whether or not *RTI Data Distribution Service* will store and deliver data that were previously published to new *DataReaders*. See Section 6.5.6.
History	Specifies how much data must to stored by *RTI Data Distribution Service*for the *DataWriter* or *DataReader*. This QosPolicy affects the RELIABILITY QosPolicy (Section 6.5.17) as well as the DURABILITY QosPolicy (Section 6.5.6). See Section 6.5.8.
LatencyBudget	Suggestion to DDS on how much time is allowed to deliver data. See Section 6.5.9.
Liveliness	Specifies and configures the mechanism that allows *DataReaders* to detect when *DataWriter*s become disconnected or "dead." See Section 6.5.11.
Property	Stores name/value (string) pairs that can be used to configure certain parameters of *RTI Data Distribution Service* 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 Section 6.5.15.
ReaderDataLifeCycle	Controls how a *DataReader* manages the lifecycle of the data that it has received. See Section 7.6.3.
Reliability	Specifies whether or not DDS will deliver data reliably. See Section 6.5.17.
ResourceLimits	Controls the amount of physical memory allocated for DDS entities, if dynamic allocations are allowed, and how they occur. Also controls memory usage among different instance values for keyed topics. See Section 6.5.18.
TimeBasedFilter	Set by a *DataReader* to limit the number of new data values received over a period of time. See Section 7.6.4.

Table 7.15 **DataReader QosPolicies**

QosPolicy	Description
TransportMulticast	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 Section 7.6.5.
TransportSelection	Allows you to select which physical transports a *DataWriter* or *DataReader* may use to send or receive its data. See Section 6.5.20.
TransportUnicast	Specifies a subset of transports and port number that can be used by an Entity to receive data. See Section 6.5.21.
TypeSupport	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 Section 6.5.22.
UserData	Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data. See Section 6.5.23.

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 existing *DataReader*, use the **get_qos()** and **set_qos()** operations on the *DataReader*. This is a general pattern for all DDS Entities, described in more detail in Section 4.1.7.3.

7.3.7.1 Configuring QoS Settings when the DataReader is Created

As described in Creating DataReaders (Section 7.3.1), 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 7.9 on page 7-30, we saw 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 *DataReader* should be used. The default DataReader QoS values are configured in the *Publisher* 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 Section 4.1.7, this is a general pattern that applies to the construction of all DDS *Entities*.

- ❑ To create a *DataReader* with non-default QoS without using a QoS Profile, see the example code in Figure 7.11 on page 7-50. It uses the *Publisher'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 7.12 on page 7-51.
- ❑ If you want to use a QoS profile, but then make some changes to the QoS before creating the *DataReader*, call **get_datareader_qos_from_profile()** and **create_datareader()** as seen in Figure 7.13 on page 7-51.

For more information, see Creating DataReaders (Section 7.3.1) and Chapter 15: Configuring QoS with XML.

Figure 7.11 **Creating a DataReader with Modified QosPolicies (not from a profile)**

```
DDS_DataReaderQos reader_qos;[1]

// initialize reader_qos with default values
subscriber->get_default_datareader_qos(reader_qos);

// make QoS changes here
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);
```

1. Note: In C, you must initialize the QoS structures before they are used, see Section 4.2.2.

Figure 7.12 **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 7.13 **Getting QoS Values from a Profile, Changing QoS Values, Creating a DataReader with Modified QoS Values**

```
DDS_DataReaderQos reader_qos;[1]

// 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 here
reader_qos.history.depth = 5;

DDSDataReader * reader = subscriber->create_datareader(
                                     topic,
                                     reader_qos,
                                     NULL, DDS_STATUS_MASK_NONE);
if (reader == NULL) {
    // handle error
}
```

1. Note: In C, you must initialize the QoS structures before they are used, see Section 4.2.2.

7. Receiving Data

7.3.7.2 Changing QoS Settings After the DataReader Has Been Created

There are 2 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 7.14. It retrieves the current values by calling the *DataReader'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 *DataReader* 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 7.15. For more information, see Chapter 15: Configuring QoS with XML.

Figure 7.14 **Changing the QoS of an Existing DataReader (without a QoS Profile)**

```
DDS_DataReaderQos reader_qos;[1]

// 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
}
```

1. For the C API, you need to use DDS_ParticipantQos_INITIALIZER or DDS_ParticipantQos_initialize(). See Special QosPolicy Handling Considerations for C (Section 4.2.2)

Figure 7.15 **Changing the QoS of an Existing DataReader with a QoS Profile**

```
retcode = datareader->set_qos_with_profile(
                   "ReaderProfileLibrary","ReaderProfile");

if (retcode != DDS_RETCODE_OK) {
   // handle error
}
```

7.3.7.3 Using a Topic's QoS to Initialize a DataWriter's QoS

Several *DataReader* QosPolicies can also be found in the QosPolicies for *Topics* (see Section 5.1.3). 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 *DataReaders* and *DataWriters* 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 QoSs. *RTI Data Distribution Service* 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 straight forward way is to get the values of policies directly from the *Topic* and use them in the policies for the *DataReader*. Figure 6.17 on page 6-56 shows an example of how to this for a *DataWriter*; the pattern applies to *DataReaders* as well.

The *Subscriber*'s **copy_from_topic_qos()** operation can be used to copy all the common policies from the *Topic* QoS to a **DataReaderQoS**, as illustrated in Figure 6.18 on page 6-57 for *DataWriters*.

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 6.19 on page 6-58 and Figure 6.20 on page 6-58 for examples involving *DataWriters*. The same pattern applies to *DataReaders*. For more information on the use and manipulation of QoS, see Section 4.1.7.

7.3.8 Navigating Relationships Among Entities

7.3.8.1 Finding Matching Publications

The following *DataReader* operations can be used to get information about the *DataWriters* that will send data to this *DataReader*.

- ❏ **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.

Note that *DataWriter* 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 Section 14.4.2 for more on **ignore_publication()**.

You can also get the DATA_READER PROTOCOL_STATUS for matching publications with **get_matched_publication_datareader_protocol_status()** (see Section 7.3.6.3).

Notes:

- Status/data for a matched publication is only kept while the matched publication is alive. Once a matched publication is no longer alive, its status is deleted. If you try to get the status/data for a matched publication that is no longer alive, the 'get data' or 'get status' call will return an error.
- The **get_matched_publication_data()** operation does not retrieve the **type_code** or **property** fields from builtin-topic data structures. This information is available through the **on_data_available()** callback (if a DataReaderListener is installed on the PublicationBuiltinTopicDataDataReader).

7.3.8.2 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.

7.3.8.3 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.

7.3.8.4 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 a **Foo** structure. For information on keys and keyed data types, please see Section 2.2.2.

7.4 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 *rtiddsgen*. Else, you will have to create them yourself, see Section 3.8.5.1 for more details.

7.4.1 Using a Type-Specific DataReader (FooDataReader)

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 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 7.8 on page 7-22 shows an example:

```
Foo_reader = FooDataReader::narrow(reader);
```

Table 7.3, "DataReader Operations," on page 7-25 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.

7.4.2 Loaning and Returning Data and SampleInfo Sequences

The **read()** and **take()** operations (and their variations) return information to your application in two sequences:

- ❑ received data samples in a sequence of the data type
- ❑ corresponding information about each 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), *RTI Data Distribution Service* will fill those sequences with memory directly loaned from the receive queue itself. There is no copying of the data or of **Sam-**

pleInfo 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* will eventually run out of memory to store data samples received from the network for that *DataReader*. See Figure 7.16 for an example of borrowing and returning loaned sequences.

```
DDS_ReturnCode_t return_loan(FooSeq &received_data,
                             DDS_SampleInfoSeq &info_seq);
```

Figure 7.16 **Using Loaned Sequences in read() and take()**

```
// In C++ and Java, sequences are automatically initialized
// to be empty
FooSeq data_seq;[1]
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);
...
```

1. 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;
```

If your code provides its own sequences to the read/take operations, then *RTI Data Distribution Service* 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 else the read/take operation will fail with a return code of **DDS_RETCODE_PRECONDITION_NOT_MET**:

- ❏ 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**.

7.4.3 Accessing Data Samples with Read or Take

To access the data samples that *RTI Data Distribution Service* has received for a *DataReader*, you must invoke the **read()** or **take()** methods. These methods return a list (sequence) of data samples and additional information about the samples in a corresponding list (sequence) of **SampleInfo** structures. The contents of **SampleInfo** are described in Section 7.4.5.

The way *RTI Data Distribution Service* builds the collection of samples depends on QoS policies set on the DataReader and Subscriber, the **source_timestamp** of the 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 *RTI Data Distribution Service* selectively returns data samples currently stored in the *DataReader's* receive queue. You may want *RTI Data Distribution Service* to return all of the data in a single list or only a subset of the available samples as configured using the **sample_states**, **view_states**, and **instance_states** masks. Section 7.4.5 describes how these masks are used to determine which data samples should be returned.

7.4.3.1 Read vs. Take

The difference between **read()** and **take()** is how *RTI Data Distribution Service* treats the data that is returned. With **take()**, *RTI Data Distribution Service* will remove the data from the *DataReader*'s receive queue. The data returned by *RTI Data Distribution Service* is no longer stored by *RTI Data Distribution Service*. With **read()**, *RTI Data Distribution Service* 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()** 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 HISTORY QosPolicy (Section 6.5.8).

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 sample, view or instance states. The only 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 ReadConditions and QueryConditions (Section 4.6.7).

As you will see, **read** and **take** have the same parameters:

```
DDS_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);

DDS_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 *RTI Data Distribution Service* memory, which must be returned with **return_loan()**. See Loaning and Returning Data and SampleInfo Sequences (Section 7.4.2).

Both operations return an ordered collection of data samples (in the **received_data_seq** parameter) and information about each sample (in the **info_seq** parameter). Exactly how they are ordered depends on the setting of the PRESENTATION QosPolicy (Section 6.4.6) and the DESTINATION_ORDER QosPolicy (Section 6.5.5). For more details please see the online 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 samples that are returned by those methods. With different combinations of these three parameters, you can direct *RTI Data Distribution Service* to return all data samples, data samples that you have not accessed before, the data samples of instances that you have not seen before, data samples of instances that have been disposed, etc. The possible values for the different states are described both in the online documentation and in Section 7.4.5.

Table 7.16 lists the variations of the **read()** and **take()** operations.

Table 7.16 **Read and Take Operations**

Read Operations	Take Operations	Description	Reference
read	take	Reads/takes a collection of data samples from the *DataReader*. Can be used for both keyed and non-keyed data types.	Section 7.4.3
read_instance	take_instance	Identical to read/take, but all samples returned belong to a single instance, which you specify as a parameter. Can only be used with keyed data types.	Section 7.4.3.4
read_next_instance	take_next_instance	Similar to read_instance, but the actual instance is not directly specified as a parameter. Instead, the samples will all belong to instance ordered after the instance that is specified by the previous_handle parameter.	Section 7.4.3.5
read_next_instance_ w_condition	take_next_instance_ w_condition	Accesses a collection of data samples of the next instance that match a specific set of *ReadConditions*, from the *DataReader*.	Section 7.4.3.7
read_next_sample	take_next_sample	These provide a convenient way to access the next data sample in the receive queue that has not been accessed before.	Section 7.4.3.3
read_w_condition	take_w_condition	Accesses a *collection* of data samples from the *DataReader* that match specific *ReadCondition* criteria.	Section 7.4.3.6

7.4.3.2 General Patterns for Accessing Data

Once the data samples are available to the data readers, the 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 PRESENTATION QosPolicy (Section 6.4.6) must be set properly.

- ❑ **Accessing 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 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.

7.4.3.3 read_next_sample and take_next_sample

The **read_next_sample()** or **take_next_sample()** operation is used to retrieve the next sample that hasn't already been accessed. It is a simple way to 'read' samples and frees your application from managing sequences and specifying 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**.

```
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 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 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.

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**.

7.4.3.4 read_instance and take_instance

The **read_instance()** and **take_instance()** operations are identical to **read()** and **take()**, but they are used to access 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 Section 2.2.2 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.

```
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,
                  DDS_ViewStateMask        view_states,
                  DDS_InstanceStateMask    instance_states);
```

Note: This operation may loan internal *RTI Data Distribution Service* memory, which must be returned with **return_loan()**. See Loaning and Returning Data and SampleInfo Sequences (Section 7.4.2).

7.4.3.5 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 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 data samples, instead you pass the handle to a 'previous' instance. The returned samples will all belong to the 'next' instance, where the ordering of instances is explained below.

```
DDS_ReturnCode_t read_next_instance(
                  FooSeq                 &received_data,
                  DDS_SampleInfoSeq      &info_seq,
                  DDS_Long                max_samples,
                  const DDS_InstanceHandle_t &previous_handle,
                  DDS_SampleStateMask     sample_states,
                  DDS_ViewStateMask       view_states,
                  DDS_InstanceStateMask   instance_states)
```

RTI Data Distribution Service orders all instances relative to each other.[1] 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.

This operation will return values for the *next* instance handle that has 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.

1. 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 *RTI Data Distribution Service* (and likely other DDS implementations as well), the order is not likely to be meaningful to you as a developer; it is simply important that some ordering exists.

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 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 though all the instances, take all the 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 in Figure 7.17 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 samples for a different instance, since the previous time through the loop, all of the samples of the previous instance were returned (and thus accessed).

Note: This operation may loan internal *RTI Data Distribution Service* memory, which must be returned with **return_loan()**. See Loaning and Returning Data and SampleInfo Sequences (Section 7.4.2).

7.4.3.6 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 *RTI Data Distribution Service* memory, which must be returned with **return_loan()**. See Loaning and Returning Data and SampleInfo Sequences (Section 7.4.2).

7.4.3.7 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

7. Receiving Data

Figure 7.17 **Using take_next_instance() to process received data**

```
FooSeq            received_data;1

DDS_SampleInfoSeq info_seq;

while (retcode = reader->take_next_instance(
                      received_data, info_seq,
                      DDS_LENGTH_UNLIMITED,
                      DDS_HANDLE_NIL,
                      DDS_ANY_SAMPLE_STATE,
                      DDS_ANY_VIEW_STATE,
                      DDS_ANY_INSTANCE_STATE)
       != DDS_RETCODE_NO_DATA) {

    // the data samples returned in received_data will all
    // 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
    }
}
```

1. 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;
```

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_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 *RTI Data Distribution Service* memory, which must be returned with **return_loan()**. See Loaning and Returning Data and SampleInfo Sequences (Section 7.4.2).

7.4.4 The Sequence Data Structure

The DDS specification uses sequences whenever a variable-length array of elements must be passed through the API. This includes passing QosPolicies into *RTI Data Distribution Service*, as well as retrieving data samples from *RTI Data Distribution Service*. 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* to copy the received data samples directly into data structures allocated in user space.

Please do not confuse (a) the user loaning memory to a sequence with (b) *RTI Data Distribution Service* 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_discontinguous()**.

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, **<*type*>Seq_initialize()**, to initialize sequences.

For C++, C++/CLI, and C# developers: 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, *RTI Data Distribution Service*'s 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**.

RTI Data Distribution Service provides a more efficient implementation for sequences of primitive types. In *RTI Data Distribution Service*, 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 online documentation (from the main page, select **Modules, Infrastructure, Sequence Support**).

7.4.5 The SampleInfo Structure

When you invoke the **read/take** operations, for every data sample that is returned, a corresponding **SampleInfo** is also returned. **SampleInfo** structures provide you with additional information about the data samples received by *RTI Data Distribution Service*.

Table 7.17 shows the format of the **SampleInfo** structure.

Table 7.17 **DDS_SampleInfo Structure**

Type	Field Name	Description
DDS_SampleStateKind	sample_state	See Section 7.4.5.2
DDS_ViewStateKind	view_state	See Section 7.4.5.3
DDS_InstanceStateKind	instance_state	See Section 7.4.5.4
DDS_Time_t	source_timestamp	Time stored by the DataWriter when the sample was written.
DDS_InstanceHandle_t	instance_handle	Handle to the data-instance corresponding to the sample.
DDS_InstanceHandle_t	publication_handle	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().
DDS_Long	disposed_generation_count	See Section 7.4.5.5.
	no_writers_generation_count	
	sample_rank	
	generation_rank	
	absolute_generation_rank	
DDS_Boolean	valid_data	Indicates whether the data sample includes valid data. See Section 7.4.5.6.
DDS_Time_t	reception_timestamp	Time stored when the sample was committed by the DataReader. See Section 7.4.5.1.
DDS_SequenceNumber_t	publication_sequence_number	Publication sequence number assigned when the sample was written by the DataWriter.
DDS_SequenceNumber_t	reception_sequence_number	Reception sequence number assigned when the sample was committed by the DataReader. See Section 7.4.5.1.

7.4.5.1 Reception Timestamp

In reliable communication, if data samples are received out received of order, *RTI Data Distribution Service* will not deliver them until all the previous data samples have been received. For example, if Sample 2 arrives before Sample 1, Sample 2 cannot be delivered until Sample 1 is received. The **reception_timestamp** is the time when all previous samples has been received—the time at which the sample is *committed*. If samples are all received in order, the committed time will be same as reception time. However, if samples are lost on the wire, then the committed time will be later than the initial reception time.

7.4.5.2 Sample States

For each sample received, *RTI Data Distribution Service* keeps a **sample_state** relative to each *DataReader*. The **sample_state** can be either:

- ❑ **READ** The *DataReader* has already accessed that sample by means of **read()**.
- ❑ **NOT_READ** The *DataReader* has never accessed that sample before.

The samples retrieved by a **read()** or **take()** need not all have the same **sample_state**.

7.4.5.3 View States

For each instance (identified by a unique key value), *RTI Data Distribution Service* 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 samples of the instance, or else the *DataReader* has accessed previous 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 Section 7.4.5.5).
- ❑ **NOT_NEW** The *DataReader* has already accessed 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 data sample). Thus all data samples related to the same instance that are returned by **read()** or **take()** will have the same value for **view_state**.

7.4.5.4 Instance States

RTI Data Distribution Service keeps an **instance_state** for each instance; it can be:

- **ALIVE** The following are all true: (a) 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 else more 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.
- **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 writing that instance.

The events that cause the **instance_state** to change can depend on the setting of the OWNERSHIP QosPolicy (Section 6.5.13):

- 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 and OWNERSHIP_STRENGTH QosPolicies. 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 data samples related to the same instance that are returned by **read()** or **take()** will have the same value for **instance_state**.

7.4.5.5 Generation Counts and Ranks

Generation counts and ranks allow your application to distinguish 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 to 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 data samples for the instance. This means that the data samples returned by **read()** and **take()** may cross generations. That is, some samples were published when the instance was alive in one generation and other 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 data samples by the generation in which they were published.

Each *DataReader* keeps two counters for each *new* 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**. The counter is reset when the instance resource is reclaimed.
- **no_writers_generation_count** Counts how many times the **instance_state** of the corresponding instance changes from **NOT_ALIVE_NO_WRITERS** to **ALIVE**. The counter is reset when the instance resource is reclaimed.

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 sample was received.

The **sample_rank** and **generation_rank** in the **SampleInfo** structure are computed relative to the sequence of samples returned by **read()** or **take()**:

- **sample_rank** Indicates how many samples of the same instance follow the current one in the sequence. The samples are always time-ordered, thus the newest sample of an instance will have a **sample_rank** of 0. Depending on what you have configured **read()** and **take()** to return, a **sample_rank** of 0 may or may not be the newest sample that was ever received. It is just the newest sample in the sequence that was returned.
- **generation_rank** Indicates the difference in 'generations' between the sample and the newest sample of the same instance as returned in the sequence. If a sample belongs to the same generation as the newest 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 sample and the newest 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 sample that belongs to the newest generation of the instance will have an **absolute_generation_rank** of 0.

Like the 'generation count' values, the 'rank' values are also reset to 0 if the instance resource is reclaimed.

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 data samples received for the instance. For example:

- A sample with **sample_rank = 0** is the newest sample of the instance in the returned sequence.
- Samples that belong to the same generation will have the same **generation_rank** (as well as **absolute_generation_rank**).
- Samples with **absolute_generation_rank = 0** belong to the newest generation for the instance received by the *DataReader*.

7.4.5.6 Valid Data Flag

The **SampleInfo** structure's **valid_data** flag indicates whether the sample contains data or is only used to communicate a change in the **instance_state** of the instance.

Normally, each sample contains both a SampleInfo structure and some data. However, there are situations in which the sample only contains the SampleInfo and does not have any associated data. This occurs when *RTI Data Distribution Service* notifies the application of a change of state for an instance that was caused by some internal mechanism (such as a timeout) for which there is no associated data. An example is when*RTI Data Distribution Service* detects that an instance has no writers and changes the corresponding **instance_state** to NOT_ALIVE_NO_WRITERS.

If this flag is TRUE, then the sample contains valid Data. If the flag is FALSE, the 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 sample, and only access the data if it is TRUE.

7.5 Subscriber QosPolicies

Subscribers have the same set of QosPolicies as *Publishers*; see Publisher/Subscriber QosPolicies (Section 6.4).

- ❏ ENTITYFACTORY QosPolicy (Section 6.4.2)
- ❏ EXCLUSIVE_AREA QosPolicy (DDS Extension) (Section 6.4.3)
- ❏ GROUP_DATA QosPolicy (Section 6.4.4)
- ❏ PARTITION QosPolicy (Section 6.4.5)
- ❏ PRESENTATION QosPolicy (Section 6.4.6)

7.6 DataReader QosPolicies

This section describes the QosPolicies that are strictly for *DataReaders* (not for *DataWriters*). For a complete list of QosPolicies that apply to *DataReaders*, see Table 7.15 on page 7-48.

- ❏ DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1)
- ❏ DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2)
- ❏ TIME_BASED_FILTER QosPolicy (Section 7.6.4)
- ❏ TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5)

7.6.1 DATA_READER_PROTOCOL QosPolicy (DDS Extension)

The DATA_READER_PROTOCOL QosPolicy applies only to *DataReaders* that are set up for reliable operation (see RELIABILITY QosPolicy (Section 6.5.17)). This policy allows the application to fine-tune the reliability protocol separately for each *DataReader*. For details of the reliable protocol used by *RTI Data Distribution Service*, see Chapter 10.

DDS has a standard protocol for packet (user and meta data) exchange between applications using DDS for communications. 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 RELIABILITY QosPolicy (Section 6.5.17) for more information on the per-DataReader/DataWriter reliability configuration. The HISTORY QosPolicy (Section 6.5.8) and RESOURCE_LIMITS QosPolicy (Section 6.5.18) also play an important role in the DDS reliability protocol.

This policy includes the members presented in Table 7.18 and Table 7.19. For defaults and valid ranges, please refer to the online documentation.

When setting the fields in this policy, the following rule applies. If this is false, *RTI Data Distribution Service* returns **DDS_RETCODE_INCONSISTENT_POLICY** when setting the QoS:

- ❑ **max_heartbeat_response_delay >= min_heartbeat_response_delay**

7.6.1.1 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 Chapter 12: Discovery.)

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

Table 7.18 **DDS_DataReaderProtocolQosPolicy**

Type	Field Name	Description
DDS_GUID_t	virtual_guid	The virtual GUID (Global Unique Identifier) is used to uniquely identify the same *DataReader* across multiple incarnations. In other words, this value allows *RTI Data Distribution Service* 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 Durability and Persistence Based on Virtual GUIDs (Section 11.2). By default, *RTI Data Distribution Service* will assign a virtual GUID automatically. If you want to restore the state of the reader after a restart, you can retrieve the value of the reader's virtual GUID using the *DataReader's* **get_qos()** operation, and set the virtual GUID of the restarted *DataReader* to the same value.
DDS_UnsignedLong	rtps_object_id	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 WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9), together with the 3 least significant bytes in rtps_object_id, and another byte assigned by *RTI Data Distribution Service* to identify the entity type, forms the BuiltinTopicKey in SubscriptionBuiltinTopicData.

Table 7.18 **DDS_DataReaderProtocolQosPolicy**

Type	Field Name	Description
DDS_Boolean	expects_inline_qos	Specifies whether this *DataReader* expects inline QoS with every sample. *DataReaders* usually rely on the discovery process to propagate QoS changes for matched *DataWriters*. Another way to get QoS information is to have it sent inline with a sample. With *RTI Data Distribution Service*, *DataWriters* and *DataReaders* 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 *DataReaders* do not cache Discovery information. The complete set of QoS that a *DataWriter* 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.
DDS_Boolean	disable_positive_acks	Determines whether the *DataReader* sends positive acknowledgements (ACKs) to matching *DataWriters*. When TRUE. the matching *DataWriter* will keep samples in its queue for this *DataReader* for a minimum keep duration (see Disabling Positive Acknowledgements (Section 6.5.2.3)). When strict-reliability is not required and NACK-based reliability is sufficient, setting this field reduces overhead network traffic.

Table 7.18 **DDS_DataReaderProtocolQosPolicy**

Type	Field Name	Description
DDS_Boolean	propagate_dispose_of_unregistered_instances	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. This field only applies to keyed readers. When set to TRUE, the *DataReader* will receive dispose notifications even if the instance is not alive. To make sure the key is available to the FooDataReader's get_key_value() operation, use this option in combination with setting the *DataWriter's* serialize_key_with_dispose field (in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)) to TRUE. See Propagating Serialized Keys with Disposed-Instance Notifications (Section 6.5.2.5).
DDS_RtpsReliableReaderProtocol_t	rtps_reliable_reader	See Table 7.19

Table 7.19 **DDS_RtpsReliableReaderProtocol_t**

Type	Field Name	Description
DDS_Duration_t	heartbeat_suppression_duration	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.
DDS_Duration_t	min_heartbeat_response_delay	Minimum delay between when the *DataReader* receives a heartbeat and when it sends an ACK/NACK.
DDS_Duration_t	max_heartbeat_response_delay	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.
DDS_Duration_t	nack_period	Rate at which to send negative acknowledgements to new *DataWriters*. See Section 7.6.1.1.

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 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.

7.6.1.2 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*.

7.6.1.3 Related QosPolicies

- DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)
- RELIABILITY QosPolicy (Section 6.5.17)

7.6.1.4 Applicable Entities

- DataReaders (Section 7.3)

7.6.1.5 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 **min_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.

7.6.2 DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension)

The DATA_READER_RESOURCE_LIMITS QosPolicy extends your control over the memory allocated by *RTI Data Distribution Service* for *DataReaders* beyond what is offered by the RESOURCE_LIMITS QosPolicy (Section 6.5.18). RESOURCE_LIMITS controls memory allocation with respect to the *DataReader* itself: the number of samples that it can store in the receive queue and the number of instances that it can manage simultaneously. DATA_READER_RESOURCE_LIMITS controls memory allocation on a per matched-*DataWriter* basis. The two are orthogonal.

This policy includes the members in Table 7.20, "DDS_DataReaderResourceLimitsQosPolicy," on page 7-79. For defaults and valid ranges, please refer to the online documentation.

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 data samples larger than the MTU of the underlying network transport); how many simultaneous outstanding loans of internal memory holding 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 to bound the amount of memory that can be used by a *DataReader*. Setting the initial size to the maximum size will prevent *RTI Data Distribution Service* 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 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.

Table 7.20 **DDS_DataReaderResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_Long	max_remote_writers	Maximum number of *DataWriters* from which a *DataReader* may receive data samples, among all instances. For unkeyed *Topics*: max_remote_writers must = max_remote_writers_per_instance
	max_remote_writers_per_instance	Maximum number of *DataWriters* from which a *DataReader* may receive data samples for a single instance. For unkeyed *Topics*: max_remote_writers must = max_remote_writers_per_instance
	max_samples_per_remote_writer	Maximum number of samples received out-of-order that a *DataReader* can store from a single reliable *DataWriter*. max_samples_per_remote_writer must be <= RESOURCE_LIMITS::max_samples
	max_infos	Maximum number of DDS_SampleInfo structures that a *DataReader* can allocate. max_infos must be >= RESOURCE_LIMITS::max_samples
	initial_remote_writers	Initial number of *DataWriters* from which a *DataReader* may receive data samples, including all instances. For unkeyed Topics: initial_remote_writers must = initial_remote_writers_per_instance
	initial_remote_writers_per_instance	Initial number of *DataWriters* from which a *DataReader* may receive data samples for a single instance. For unkeyed *Topics*: initial_remote_writers must = initial_remote_writers_per_instance
	initial_infos	Initial number of DDS_SampleInfo structures that a *DataReader* will allocate.
	initial_outstanding_reads	Initial number of times in which memory can be concurrently loaned via read/take calls without being returned with return_loan().
	max_outstanding_reads	Maximum number of times in which memory can be concurrently loaned via read/take calls without being returned with return_loan().
	max_samples_per_read	Maximum number of samples that can be read/taken on a *DataReader*.
DDS_Boolean	disable_fragmentation_support	Determines whether the *DataReader* can receive fragmented samples. When fragmentation support is not needed, disabling fragmentation support will save some memory resources.

7. Receiving Data

Table 7.20 **DDS_DataReaderResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_ Long	max_fragmented_ samples	The maximum number of 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 samples while waiting for the remaining fragments. These samples need not have consecutive sequence numbers and may have been sent by different *DataWriters*. Once all fragments of a sample have been received, the sample is treated as a regular sample and becomes subject to standard QoS settings, such as max_samples. *RTI Data Distribution Service* will drop fragments if the max_fragmented_samples limit has been reached. For best-effort communication, *RTI Data Distribution Service* will accept a fragment for a new sample, but drop the oldest fragmented sample from the same remote writer. For reliable communication, *RTI Data Distribution Service* will drop fragments for any new samples until all fragments for at least one older sample from that writer have been received. Only applies if disable_fragmentation_support is DDS_BOOLEAN_FALSE.
	initial_fragmented_ samples	The initial number of samples for which a *DataReader* may store fragments. Only applies if disable_fragmentation_support is DDS_BOOLEAN_FALSE.
	max_fragmented_ samples_per_remote_ writer	The maximum number of 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 DDS_BOOLEAN_FALSE.
	max_fragments_per_ sample	Maximum number of fragments for a single sample. Only applies if disable_fragmentation_support is DDS_BOOLEAN_FALSE.

Table 7.20 **DDS_DataReaderResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_ Boolean	dynamically_allocate_ fragmented_samples	Determines whether the *DataReader* pre-allocates storage for storing fragmented samples. By default, *RTI Data Distribution Service* 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. If dynamically_allocate_fragmented_samples is set to DDS_BOOLEAN_TRUE, *RTI Data Distribution Service* does not allocate memory up front, 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 may be 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 *RTI Data Distribution Service* will have sufficient resources at run-time. Also, dynamic memory allocation and memory freeing at run time may not give you good performance. Only applies if disable_fragmentation_support is DDS_BOOLEAN_FALSE.
DDS_ Long	max_total_instances	Maximum number of instances for which a *DataReader* will keep state. See max_total_instances and max_instances (Section 7.6.2.1)
DDS_ Long	max_remote_virtual_ writers_per_instance	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 *RTI Persistence Service*[a], require *RTI Data Distribution Service* to keep some internal state per virtual writer and instance that is used to filter duplicate samples. These duplicate samples could be coming from different *DataWriter* channels or from multiple executions of *RTI 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, *RTI Data Distribution Service* will not associate this instance with new virtual writers. Duplicates samples coming from these virtual writers will not be filtered on the reader. If you are not using Durable Reader State, MultiChannel DataWriters or *RTI Persistence Service*, you can set this property to 1 to optimize resources. For additional information about the virtual writers see Chapter 11.

Table 7.20 **DDS_DataReaderResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_Long	initial_remote_virtual_writers_per_instance	Initial number of virtual remote writers per instance. For unkeyed types, this value is ignored.
DDS_Long	max_query_condition_filters	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.

a. *RTI Persistence Service* is included with *RTI Data Distribution Service Professional Edition* and *Elite Edition*. It saves data samples so they can be delivered to subscribing applications that join the system at a later time (see Chapter 21: Introduction to RTI Persistence Service).

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 *RTI Data Distribution Service* to support a RELIABLE setting for the RELIABILITY QosPolicy (Section 6.5.17), the *DataReader* must temporarily store data samples that have been received out-of-order from a reliable *DataWriter*. The storage of out-of-order 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 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 RESOURCE_LIMITS QosPolicy (Section 6.5.18).

max_samples_per_remote_writer allows *RTI Data Distribution Service* 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 data samples are being resent.

When setting the values of the members, the following rules apply:

- **max_remote_writers >= initial_remote_writers**
- **max_remote_writers_per_instance >= initial_remote_writers_per_instance**
  max_remote_writers_per_instance <= max_remote_writers
- **max_infos >= initial_infos**
  max_infos >= RESOURCE_LIMITS::max_samples

- ❑ **max_outstanding_reads >= initial_outstanding_reads**
- ❑ **max_remote_writers >= max_remote_writers_per_instance**
- ❑ **max_samples_per_remote_writer <= RESOURCE_LIMITS::max_samples**

If any of the above are false, *RTI Data Distribution Service* returns the error code **DDS_RETCODE_INCONSISTENT_POLICY** when setting the *DataReader*'s QoS.

7.6.2.1 max_total_instances and max_instances

The maximum number of instances actively managed by a *DataReader* is determined by **max_instances** in the RESOURCE_LIMITS QosPolicy (Section 6.5.18). These instances have associated *DataWriters* or samples in the *DataReader's* queue and are visible to the user through operations such as **take()**, **read()**, and **get_key()**.

The features Durable Reader State (Section 11.4), multi-channel *DataWriters* (Chapter 16), and *RTI Persistence Service*[1] require *RTI Data Distribution Service* to keep some internal state even for instances without *DataWriters* or samples in the *DataReader's* queue. The additional state is used to filter duplicate samples that could be coming from different *DataWriter* channels or from multiple executions of *RTI Persistence Service*. The total maximum number of instances that will be managed by the middleware, including instances without associated *DataWriters* or samples, is determined by **max_total_instances**.

max_total_instances must be greater than **max_instances** or equal to DDS_AUTO_MAX_TOTAL_INSTANCES, which treats **max_total_instances** as being equal to **max_instances** in the RESOURCE_LIMITS QosPolicy (Section 6.5.18).

When a new instance is received, *RTI Data Distribution Service* will check the resource limit **max_instances** in the RESOURCE_LIMITS QosPolicy (Section 6.5.18). If the limit is exceeded, *RTI Data Distribution Service* will drop the sample and report it as lost and rejected. If the limit is not exceeded, *RTI Data Distribution Service* will check **max_total_instances**. If **max_total_instances** is exceeded, *RTI Data Distribution Service* will replace an existing instance without DataWriters and samples with the new one. The application could receive duplicate samples for the replaced instance if it becomes alive again.

7.6.2.2 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

1. *RTI Persistence Service* is included with *RTI Data Distribution Service Professional Edition* and *Elite Edition*. It saves data samples so they can be delivered to subscribing applications that join the system at a later time (see Chapter 21: Introduction to RTI Persistence Service).

QosPolicy's **max_samples** to prevent a single *DataWriter* from starving others. This control is especially important for *Topics* that have their OWNERSHIP QosPolicy (Section 6.5.13) 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.

7.6.2.3 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*.

7.6.2.4 Related QosPolicies

- ❏ RESOURCE_LIMITS QosPolicy (Section 6.5.18)
- ❏ OWNERSHIP QosPolicy (Section 6.5.13)

7.6.2.5 Applicable Entities

- ❏ DataReaders (Section 7.3)

7.6.2.6 System Resource Considerations

Increasing any of the "initial" values in this policy will increase the amount of memory allocated by *RTI Data Distribution Service* 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, the *RTI Data Distribution Service* may dynamically allocate memory in response to network communications.

7.6.3 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 *RTI Data Distribution Service* will automatically remove data from the receive queue (so that user applications cannot access it afterwards) when *RTI Data Distribution Service* 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 *RTI Data Distribution Service* automatically removes disposed data from the receive queue.

For keyed Topics, the consideration of removing data samples from the receive queue is done on a per instance (key) basis. Thus when *RTI Data Distribution Service* 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 data samples for a certain instance (key). *DataWriters* also can dispose its data on a per instance basis. Only the data samples of disposed instances would be removed by *RTI Data Distribution Service* if so configured.

This policy helps purge untaken samples from not-alive-instances and thus may prevent a *DataReader* from reclaiming resources. With this policy, the untaken samples from not-alive-instances are purged and treated as if the samples were taken after the specified amount of time.

The *DataReader* internally maintains the samples that have not been taken by the application, subject to the constraints imposed by other QoS policies such as HISTORY QosPolicy (Section 6.5.8) and RESOURCE_LIMITS QosPolicy (Section 6.5.18).

The *DataReader* also maintains information regarding the identity, view-state, and instance-state of data instances, even after all samples have been 'taken' (see Section 7.4.3). This is needed to properly compute the states when future samples arrive.

Under normal circumstances, a *DataReader* can only reclaim all resources for instances for which there are no *DataWriters* and for which all samples have been 'taken.' The last 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 samples for some reason. The 'untaken' samples will prevent the *DataReader* from reclaiming the resources and they would remain in the *DataReader* indefinitely.

It includes the members in Table 7.21.

Table 7.21 **DDS_ReaderDataLifecycleQosPolicy**

Type	Field Name	Description
DDS_Duration_t	autopurge_nowriter_ samples_delay	How long the *DataReader* maintains information about an instance once its instance_state becomes NOT_ALIVE_NO_WRITERS.
DDS_Duration_t	autopurge_disposed_ samples_delay	How long the *DataReader* maintains information about an instance once its instance_state becomes NOT_ALIVE_DISPOSED.

- ❏ **autopurge_nowriter_samples_delay** This defines the maximum duration for which the *DataReader* will maintain information regarding an instance once its **instance_state** becomes **NOT_ALIVE_NO_WRITERS**. After this time elapses, the *DataReader* will purge all internal information regarding the instance, any untaken samples will also be lost.
- ❏ **autopurge_disposed_samples_delay** This defines the maximum duration for which the *DataReader* will maintain samples of an instance once its **instance_state** becomes **NOT_ALIVE_DISPOSED**. After this time elapses, the *DataReader* will purge all internal information regarding the instance; any untaken samples will also be lost.

7.6.3.1 Properties

This QoS policy *can* be modified after the *DataReader* is enabled.

It only applies to *DataReaders*, so there are no RxO restrictions for setting it compatibly on the *DataWriter*.

7.6.3.2 Related QoS Policies

- ❏ HISTORY QosPolicy (Section 6.5.8)
- ❏ LIVELINESS QosPolicy (Section 6.5.11)
- ❏ OWNERSHIP QosPolicy (Section 6.5.13)
- ❏ RESOURCE_LIMITS QosPolicy (Section 6.5.18)
- ❏ WRITER_DATA_LIFECYCLE QoS Policy (Section 6.5.24)

7.6.3.3 Applicable Entities

- ❑ DataReaders (Section 7.3)

7.6.3.4 System Resource Considerations

None.

7.6.4 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 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.

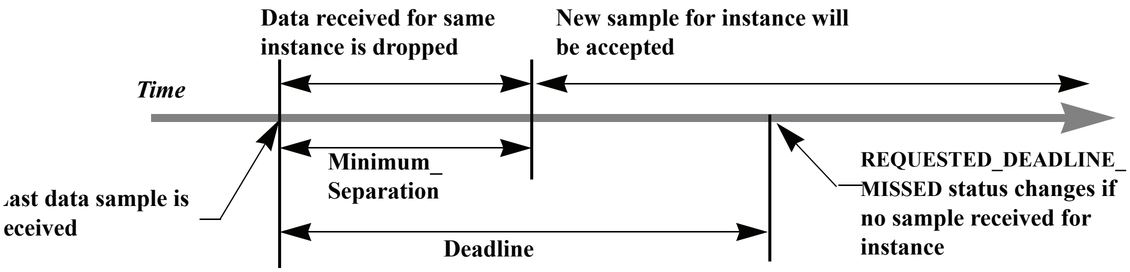
It includes the member in Table 7.22. For the default and valid range, please refer to the online documentation.

Table 7.22 **DDS_TimeBasedFilterQosPolicy**

Type	Field Name	Description
DDS_Duration_t	minimum_separation	Minimum separation time between samples of the same instance. Must be <= DEADLINE::period

As seen in Figure 7.18, it is inconsistent to set a *DataReader's* **minimum_separation** longer than its DEADLINE QosPolicy (Section 6.5.4) **period**.

Figure 7.18 **Accepting Data for DataReaders**



Data samples for a DataReader *can be filtered out using the TIME_BASED_FILTER QoS (**minimum_separation**). Once a data sample for an instance has been received, RTI Data Distribution Service 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 sample that arrives is accepted and stored in the receive queue, and the timer starts again. If no 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 samples for a data instance to be received at a specified period, then there is no need for *RTI Data Distribution Service* 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, unicast data delivery, if the data type is unkeyed and the *DataWriter* has an infinite liveliness **lease_duration** (LIVELINESS QosPolicy (Section 6.5.11)), *RTI Data Distribution Service* 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.

Other configurations (for example, when the *DataWriter* is reliable, or the data type is keyed, or the *DataWriter* has finite a 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' samples (that is, samples that have *not* been disposed/unregistered).

7.6.4.1 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 *RTI Data Distribution Service* can also be reduced thus minimizing the consumption of network bandwidth.

You may want to change the **minimum_separation** between 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 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 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.

7.6.4.2 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*.

7.6.4.3 Related QosPolicies

- ❏ RELIABILITY QosPolicy (Section 6.5.17)
- ❏ DEADLINE QosPolicy (Section 6.5.4)
- ❏ TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5)

7.6.4.4 Applicable Entities

- ❏ DataReaders (Section 7.3)

7.6.4.5 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.

7.6.5 TRANSPORT_MULTICAST QosPolicy (DDS Extension)

This QosPolicy specifies the multicast address on which a *DataReader* 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 DDS topic.

The QosPolicy structure includes the members in Table 7.23.

Table 7.23 **DDS_TransportMulticastQosPolicy**

Type	Field Name	Description
DDS_TransportMulticastSettingSeq (A sequence of the type shown in Table 7.24)	value	A sequence of multicast locators. (See Locator Format (Section 12.2.1.1).)

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 *RTI Data Distribution Service* to automatically use a default value. As

Table 7.24 **DDS_TransportMulticastSetting_t**

Type	Field Name	Description
DDS_StringSeq	transports	A sequence of transport aliases that specifies which transports should be used to receive multicast messages for this *DataReader*.
char *	receive_address	A multicast group address to which the *DataWriter* should send data for this *DataReader*.
DDS_Long	receive_port	The port that should be used in the addressing of multicast messages destined for this *DataReader*. A value of 0 will cause *RTI Data Distribution Service* to use a default port number based on domain id. See Ports Used for Discovery (Section 12.5).

explained in Ports Used for Discovery (Section 12.5), 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, *RTI Data Distribution Service* may create a thread to process messages received for that port on that transport. See Chapter 17: RTI Data Distribution Service Threading Model 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*, *RTI Data Distribution Service* currently only uses one multicast address (the first in the sequence) per *DataReader*.

❑ If a *DataWriter* is using the MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12), the multicast addresses specified in the TRANSPORT_MULTICAST QosPolicy are ignored by that *DataWriter*. The *DataWriter* will not publish samples on those locators.

7.6.5.1 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 7.19 shows an example of how to set this QosPolicy.

Figure 7.19 **Setting Up a Multicast DataReader**

```
...

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);
```

7.6.5.2 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.

7.6.5.3 Related QosPolicies

- ❏ MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12)
- ❏ TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21)
- ❏ TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)

7.6.5.4 Applicable Entities

- ❏ DomainParticipants (Section 8.3)
- ❏ DataReaders (Section 7.3)

7.6.5.5 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, *RTI Data Distribution Service* 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 Receive Threads (Section 17.3) for more information.

Chapter 8 Working with Domains

This chapter discusses how to use *DomainParticipant*s. It describes the types of operations that are available for them and their QosPolicies.

This chapter includes the following sections:

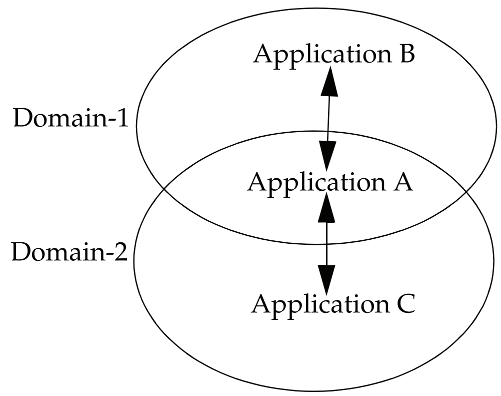
- ❑ Fundamentals of Domains and DomainParticipants (Section 8.1)
- ❑ DomainParticipantFactory (Section 8.2)
- ❑ DomainParticipants (Section 8.3)
- ❑ DomainParticipantFactory QosPolicies (Section 8.4)
- ❑ DomainParticipant QosPolicies (Section 8.5)
- ❑ Clock Selection (Section 8.6)

The goal of this chapter is to help you become familiar with the objects you need for setting up your *RTI Data Distribution Service* application. For specific details on any mentioned operations, see the online documentation.

8.1 Fundamentals of Domains and DomainParticipants

DomainParticipants are the focal point for creating, destroying, and managing other *RTI Data Distribution Service* objects. A DDS domain is a logical network of applications: only applications that belong to the same DDS domain may communicate using *RTI Data Distribution Service*. A domain is identified by a unique integer value known as a domain ID. An application participates in a domain by creating a *DomainParticipant* for that domain ID.

Figure 8.1 **Relationship between Applications and Domains**



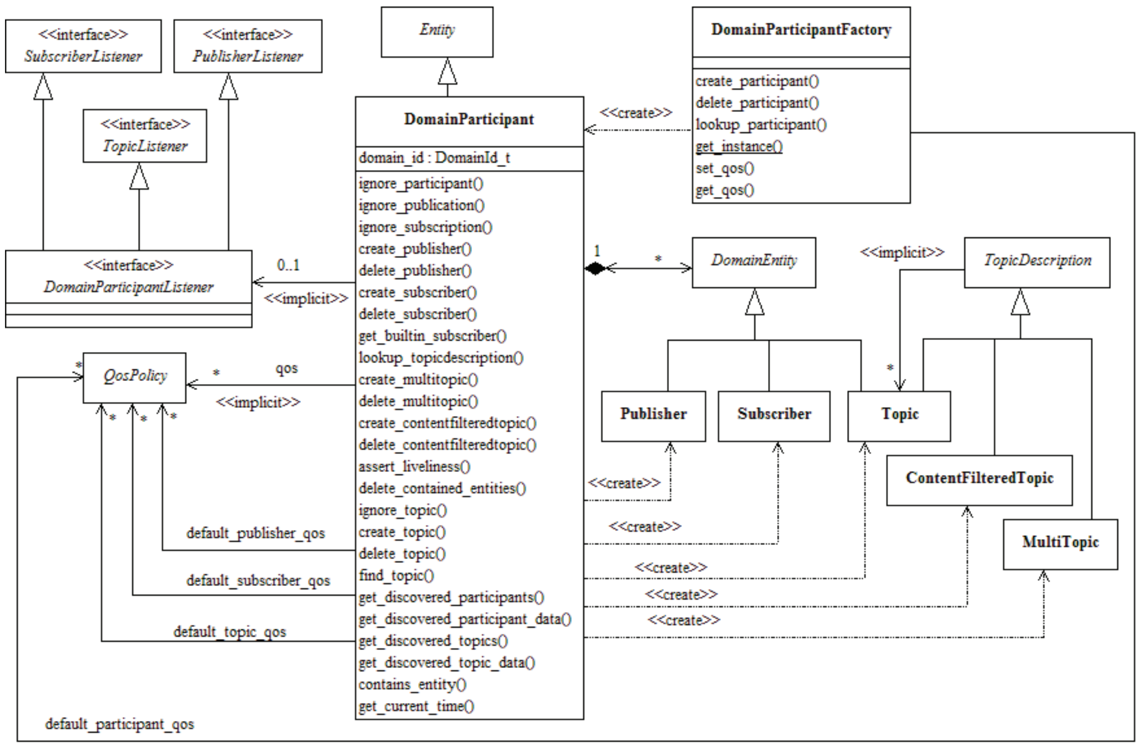
Applications can belong to multiple domains—A belongs to domains 1 and 2. Applications in the same domain can communicate with each other, such as A and B, or A and C. Applications in different domains, such as B and C, are not even aware of each other and will not exchange messages.

As seen in Figure 8.1, a single application can participate in multiple domains by creating multiple *DomainParticipants* with different domain IDs. *DomainParticipant*s in the same domain form a logical network; they are isolated from *DomainParticipant*s of other domains, even those running on the same set of physical computers sharing the same physical network. *DomainParticipants* in different domains will never exchange messages with each other. Thus, a domain establishes a “virtual network” linking all *DomainParticipant*s that share the same domain ID.

An application that wants to participate in a certain domain will need to create a "Domain Participant." As seen in Figure 8.2, a *DomainParticipant* object is a container for all other DDS *Entities* that belong to the same domain. It acts as factory for the *Publisher*, *Subscriber*, and *Topic* entities. (As seen in Chapters 6 and 7, in turn, *Publishers* are factories for *DataWriters* and *Subscribers* are factories for *DataReaders*.) *DomainParticipants* cannot contain other *DomainParticipants*.

Like all DDS *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*).

Figure 8.2 **Domain Module**



Note: MultiTopics are not currently supported.

8.2 DomainParticipantFactory

The main purpose of a *DomainParticipantFactory* is to create and destroy *DomainParticipant*s.

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 8.3 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**,[1] can also be used to retrieve the singleton factory.

Unlike the other DDS *Entities* that you create, the *DomainParticipantFactory* does not have an associated *Listener*. However, it does have associated QosPolicies, see Section 8.2.1. 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 Section 8.3.6.4.

Figure 8.3 **Instantiating a DomainParticipantFactory**

```
DDSDomainParticipantFactory* factory = NULL;

factory = DDSDomainParticipantFactory::get_instance();

if (factory == NULL) {
     // ... error
}
```

Once you have a *DomainParticipantFactory*, you can use it to perform the operations listed in Table 8.1. The most important one is **create_participant()**, described in Section 8.3.1. For more details on all operations, see the online documentation as well as the section of the manual listed in the Reference column.

1. In C, the macro is **DDS_TheParticipantFactory**. In Java, use the static class method **DomainParticipantFactory.TheParticipantFactory**.

Table 8.1 **DomainParticipantFactory Operations**

Working with ...	Operation	Description	Reference
Domain-Participants	create_participant	Creates a *DomainParticipant*.	Section 8.3.1
	create_participant_with_profile	Creates a *DomainParticipant* based on a QoS profile.	
	delete_participant	Deletes a *DomainParticipant*.	Section 8.3.2
	get_default_participant_qos	Gets the default QoS for *DomainParticipants*.	Section 8.2.2
	lookup_participant	Finds a specific *DomainParticipant*, based on a domain ID.	Section 8.2.4
	set_default_participant_qos	Sets the default QoS for *DomainParticipants*.	Section 8.2.2
	set_default_participant_qos_with_profile	Sets the default QoS for *DomainParticipants* based on a QoS profile.	
The Factory's Instance	get_instance	Gets the singleton instance of this class.	Section 8.2.3
	finalize_instance	Destroys the singleton instance of this class.	
The Factory's Own QoS	get_qos	Gets/sets the DomainParticipantFactory's QoS.	Section 4.1.7
	set_qos		

8. Domains

Table 8.1 **DomainParticipantFactory Operations**

Working with ...	Operation	Description	Reference
Profiles & Libraries	get_default_library	Gets the default library for a *DomainParticipantFactory*.	Section 8.2.1.1
	get_default_profile	Gets the default QoS profile for a *DomainParticipantFactory*.	
	get_default_profile_library	Gets the library that contains the default QoS profile for a *DomainParticipantFactory*.	
	get_<*entity*>_qos_from_profile	Gets the <*entity*> QoS values associated with a specified QoS profile. <*entity*> may be *topic*, *datareader*, *datawriter*, *subscriber*, *publisher*, or *participant*.	Section 8.2.5
	get_<*entity*>_qos_from_profile_w_topic_name	Like get_<*entity*>_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. <*entity*> may be *topic*, *datareader*, or *datawriter*.	
	get_qos_profiles	Gets the names of all XML QoS profiles associated with a specified XML QoS profile library.	Section 15.8.5
	get_qos_profile_libraries	Gets the names of all XML QoS profile libraries associated with the DomainParticipantFactory.	Section 15.9.1
	load_profiles	Explicitly loads or reloads the QoS profiles.	Section 15.2.1
	reload_profiles		
	set_default_profile	Sets the default QoS profile for a DomainParticipantFactory.	Section 8.2.1.1
	set_default_library	Sets the default library for a DomainParticipantFactory.	
	unload_profiles	Frees the resources associated with loading QoS profiles.	Section 15.2.1

8.2.1 Setting DomainParticipantFactory QosPolicies

The **DDS_DomainParticipantFactoryQos** structure has the following format:

```
struct DDS_DomainParticipantFactoryQos {
    DDS_EntityFactoryQosPolicy          entity_factory;
    DDS_SystemResourceLimitsQosPolicy   resource_limits;
    DDS_ProfileQosPolicy                profile;
};
```

For information on *why* you would want to change a particular QosPolicy, see the section referenced in Table 8.2.

Table 8.2 **DomainParticipantFactory QoS**

QosPolicy	Description
EntityFactory	Controls whether or not child entities are created in the enabled state. See Section 6.4.2.
Profile	Configures the way that XML documents containing QoS profiles are loaded by RTI. See Section 8.4.1.
SystemResource-Limits	Configures *DomainParticipant*-independent resources used by *RTI Data Distribution Service*. Mainly used to change the maximum number of *DomainParticipants* that can be created within a single process (address space). See Section 8.4.2.

8.2.1.1 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 Chapter 15: Configuring QoS with XML.

```
virtual const char *  get_default_library ()
const char *  get_default_profile ()
const char *  get_default_profile_library ()
```

There are also operations for setting the DomainParticipantFactory's default library and profile:

```
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.)

set_default_profile() 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 Section 8.2.2).

8.2.2 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 else, 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().**

8.2.3 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 DDS 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.

8.2.4 Looking Up a DomainParticipant

The DomainParticipantFactory has a useful operation for retrieving the handle to a particular *DomainParticipant*:

```
DDSDomainParticipant* lookup_participant (DDS_DomainId_t domainId)
```

8.2.5 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 6.5 on page 6-12.

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 Section 15.8.3). 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.

```
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.

8.3 DomainParticipants

A *DomainParticipant* is a container for DDS *Entity* objects that all belong to the same domain. Each *DomainParticipant* has its own set of internal threads and internal data structures that maintain information about the DDS *Entities* created by itself and other *DomainParticipants* in the same 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 8.3. For more details on all operations, see the online 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 Restricted Operations in Listener Callbacks (Section 4.5.1).

Table 8.3 **DomainParticipant Operations**

Working with ...	Operation	Description	Reference
Builtin Subscriber	get_builtin_subscriber	Returns the builtin Subscriber.	Section 14.2
Domain-Participants	add_peer	Adds an entry to the peer list.	Section 8.5.2.3
	enable	Enables the *DomainParticipant*.	Section 4.1.2
	get_discovered_participant_data	Provides the ParticipantBuiltinTopicData for a discovered *DomainParticipant*.	Section 8.3.11
	get_discovered_participants	Provides a list of *DomainParticipants* that have been discovered.	
	get_domain_id	Gets the domain ID of the *DomainParticipant*.	Section 8.3.4
	get_listener	Gets the currently installed *DomainParticipantListener*.	Section 8.3.5
	get_qos	Gets the *DomainParticipant* QoS.	Section 8.3.6
	ignore_participant	Rejects the connection to a remote *DomainParticipant*.	Section 14.4
	set_listener	Replaces the *DomainParticipantListener*.	Section 8.3.5
	set_qos	Sets the *DomainParticipant* QoS.	Section 8.3.6
	set_qos_with_profile	Sets the *DomainParticipant* QoS based on a QoS profile.	
Content-Filtered-Topics	create_contentfilteredtopic	Creates a ContentFilteredTopic that can be used to process content-based subscriptions.	Section 5.4.3
	create_contentfilteredtopic_with_filter		
	delete_contentfilteredtopic	Deletes a ContentFilteredTopic.	Section 5.4.4
	register_contentfilter	Registers a new content filter.	Section 5.4.8.1
	unregister_contentfilter	Unregisters a new content filter.	Section 5.4.8.2
	lookup_contentfilter	Gets a previously registered content filter.	Section 5.4.8.3

Table 8.3 **DomainParticipant Operations**

Working with ...	Operation	Description	Reference
DataReaders	create_datareader	Creates a *DataReader* with a given DataReaderListener, and an implicit *Subscriber.*	Section 7.3.1
	create_datareader_with_profile	Creates a *DataReader* based on a QoS profile, with a given DataReaderListener, and an implicit *Subscriber.*	
	delete_datareader	Deletes a *DataReader* that belongs to the 'implicit *Subscriber.*'	Section 7.3.2
	get_default_datareader_qos	Copies the default DataReaderQoS values into the provided structure.	Section 8.3.6.4
	ignore_subscription	Rejects the connection to a *DataReader*	
	set_default_datareader_qos	Sets the default DataReaderQos values.	
	set_default_datareader_qos_with_profile	Sets the default DataReaderQos using values from a QoS profile.	
DataWriters	create_datawriter	Creates a *DataWriter* with a given DataWriterListener, and an implicit *Publisher.*	Section 6.2.2
	create_datawriter_with_profile	Creates a *DataWriter* based on a QoS profile, with a given DataWriterListener, and an implicit *Publisher.*	
	delete_datawriter	Deletes a *DataWriter* that belongs to the 'implicit *Publisher.*'	Section 6.2.3
	ignore_publication	Rejects the connection to a *DataWriter.*	Section 14.4
	get_default_datawriter_qos	Copies the default DataWriterQos values into the provided DataWriterQos structure.	Section 8.3.6.4
	set_default_datawriter_qos	Sets the default DataWriterQoS values.	
	set_default_datawriter_qos _with_profile	Sets the default DataWriterQos using values from a profile.	

Table 8.3 **DomainParticipant Operations**

Working with ...	Operation	Description	Reference
Publishers	create_publisher	Creates a *Publisher* and a PublisherListener.	Section 6.2.2
	create_publisher_with_profile	Creates a *Publisher* based on a QoS profile, and a PublisherListener.	
	delete_publisher	Deletes a *Publisher*.	Section 6.2.3
	get_default_publisher_qos	Copies the default PublisherQos values into the provided PublisherQos structure.	Section 8.3.6.4
	get_implicit_publisher	Gets the *Publisher* that is implicitly created by the *DomainParticipant*.	Section 8.3.9
	get_publishers	Provides a list of all *Publishers* owned by the *DomainParticipant*.	Section 8.3.13.3
	set_default_publisher_qos	Sets the default PublisherQos values.	Section 8.3.6.4
	set_default_publisher_qos_with_profile	Sets the default PublisherQos using values from a QoS profile.	
Subscribers	create_subscriber	Creates a *Subscriber* and a SubscriberListener.	Section 7.2.2
	create_subscriber_with_profile	Creates a *Subscriber* based on a QoS profile, and a SubscriberListener.	
	delete_subscriber	Deletes a *Subscriber*.	Section 7.2.3
	get_default_subscriber_qos	Copies the default SubscriberQos values into the provided SubscriberQos structure.	Section 8.3.6.4
	get_implicit_subscriber	Gets the *Subscriber* that is implicitly created by the *DomainParticipant*.	Section 8.3.9
	get_subscribers	Provides a list of all *Subscribers* owned by the *DomainParticipant*.	Section 8.3.13.3
	set_default_subscriber_qos	Sets the default SubscriberQos values.	Section 8.3.6.4
	set_default_subscriber_qos_with_profile	Sets the default SubscriberQos values using values from a QoS profile.	

Table 8.3 **DomainParticipant Operations**

Working with ...	Operation	Description	Reference
Topics	create_topic	Creates a *Topic* and a TopicListener.	Section 5.1.1
	create_topic _with_profile	Creates a Topic based on a QoS profile, and a TopicListener.	
	delete_topic	Deletes a *Topic*.	
	get_default_topic_qos	Copies the default TopicQos values into the provided TopicQos structure.	Section 8.3.6.4
	get_discovered_topic_data	Retrieves the BuiltinTopicData for a discovered *Topic*.	Section 8.3.12
	get_discovered_topics	Returns a list of all (non-ignored) discovered *Topics*.	
	ignore_topic	Rejects a remote topic.	Section 14.4
	lookup_topicdescription	Gets an existing locally-created TopicDescription (Topic).	Section 8.3.7
	set_default_topic_qos	Sets the default TopicQos values.	Section 8.3.6.4
	set_default_topic_qos_with _profile	Sets the default TopicQos values using values from a profile.	
	find_topic	Finds an existing Topic, based on its name.	Section 8.3.8
Flow-Controllers	create_flowcontroller	Creates a custom FlowController object.	Section 6.6.4
	delete_flowcontroller	Deletes a custom FlowController object.	
	get_default_flowcontroller_ property	Gets the default properties used when a new FlowController is created.	Section 6.6.5
	set_default_flowcontroller_ property	Sets the default properties used when a new FlowController is created.	
	lookup_flowcontroller	Finds a FlowController, based on its name.	Section 6.6.8
Libraries and Profiles	get_default_library	Gets the default library.	Section 8.3.6.3
	get_default_profile	Gets the default profile.	
	get_default_profile_library	Gets the library that contains the default profile.	
	set_default_profile	Sets the default QoS profile.	
	set_default_library	Sets the default library.	

Table 8.3 **DomainParticipant Operations**

Working with ...	Operation	Description	Reference
MultiTopics	create_multitopic	Creates a *MultiTopic* that can be used to subscribe to multiple topics and combine/filter the received data into a resulting type.	*Currently not supported.*
	delete_multitopic	Deletes a *MultiTopic*.	
Other	assert_liveliness	Manually asserts the liveliness of this *DomainParticipant*.	Section 8.3.9
	delete_contained_entities	Recursively deletes all the entities that were created using the "create" operations on the *DomainParticipant* and its children.	Section 8.3.3
	contains_entity	Confirms if an entity belongs to the *DomainParticipant* or not.	Section 8.3.13.1
	get_current_time	Gets the current time used by *RTI Data Distribution Service*.	Section 8.3.13.2
	get_status_changes	Gets a list of statuses that have changed since the last time the application read the status or the *Listeners* were called.	Section 4.1.4

8.3.1 Creating a DomainParticipant

Typically, you will only need to create one *DomainParticipant* per domain per application. (Although unusual, you can create multiple *DomainParticipants* for the same domain in an application.)

To create a *DomainParticipant*, use the *DomainParticipantFactory's* **create_participant()** or **create_participant_with_profile()** operation:

```
DDSDomainParticipant * create_participant(
                           DDS_DomainId_t domainId,
                           const DDS_DomainParticipantQos &qos,
                           DDSDomainParticipantListener *listener,
                           DDS_StatusMask mask)

DDSDomainParticipant * create_participant_with_profile (
                           DDS_DomainId_t domainId,
                           const char * library_name,
                           const char *profile_name,
                           DDSDomainParticipantListener *listener,
                           DDS_StatusMask mask)
```

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 Chapter 15: Configuring QoS with XML.

domainId The domain ID uniquely identifies the domain that the *DomainParticipant* is in. It controls with which other *DomainParticipants* it will communicate. See Section 8.3.4 for more information on domain IDs.

qos If you want the default QoS settings (described in the online documentation), use **DDS_PARTICIPANT_QOS_DEFAULT** for this parameter (see Figure 8.4 on page 8-17). If you want to customize any of the QosPolicies, supply a **DomainParticipantQos** structure that is described in Section 8.3.6.

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. *RTI Data Distribution Service* 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 Setting Up DomainParticipantListeners (Section 8.3.5).

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 Listeners (Section 4.4).

library_name A QoS Library is a named set of QoS profiles. See QoS Libraries (Section 15.9).

profile_name A QoS profile groups a set of related QoS, usually one per entity. See QoS Profiles (Section 15.8).

After you create a *DomainParticipant*, the next step is to register the data types that will be used by the application, see Using rtiddsgen (Section 3.6). Then you will need to create the *Topics* that the application will publish and/or subscribe, see Creating Topics (Section 5.1.1). Finally, you will use the *DomainParticipant* to create *Publishers* and/or *Subscribers*, see Creating Publishers (Section 6.2.2) and Creating Subscribers (Section 7.2.2).

Note: It is not safe to create one *DomainParticipant* while another thread may simultaneously be looking up (Section 8.2.4) or deleting (Section 8.3.2) the same *DomainParticipant*.

For more examples, see Configuring QoS Settings when the DomainParticipant is Created (Section 8.3.6.1).

Figure 8.4 **Creating a DomainParticipant with Default QosPolicies**

```
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(domain_id,
                                      DDS_PARTICIPANT_QOS_DEFAULT,
                                      participant_listener,
                                      DDS_STATUS_MASK_ALL);

if (participant == NULL) {
    // ... error
};
```

8.3.2 Deleting DomainParticipants

If the application is no longer interested in communicating in a certain 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 Deleting Contained Entities (Section 8.3.3)).

To delete a *DomainParticipant*:

1. 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 (Section 8.3.3) to delete them all at the same time.

```
DDS_ReturnCode_t delete_publisher  (DDSPublisher  *p)
DDS_ReturnCode_t delete_subscriber (DDSSubscriber *s)
DDS_ReturnCode_t delete_contentfilteredtopic
               (DDSContentFilteredTopic *a_contentfilteredtopic)
DDS_ReturnCode_t delete_topic (DDSTopic *topic)
```

2. Delete the *DomainParticipant* by using the *DomainParticipantFactory's* **delete_participant()** operation.

```
DDS_ReturnCode_t delete_participant
                            (DDSDomainParticipant *a_participant)
```

Note: A *DomainParticipant* cannot be deleted within its *Listener* callback, see Restricted Operations in Listener Callbacks (Section 4.5.1).

After a *DomainParticipant* has been deleted, all of the participant's internal *RTI Data Distribution Service* threads and allocated memory will have been deleted. You should delete the *DomainParticipantListener* only after the *DomainParticipant* itself has been deleted.

8.3.3 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*.

```
DDS_ReturnCode_t delete_contained_entities( )
```

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. 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 Deleting DomainParticipants (Section 8.3.2)).

8.3.4 Choosing a Domain ID and Creating Multiple Domains

A domain ID identifies the 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 (Section 8.5.9.3). Please see the online (HTML) documentation for the DDS_RtpsWellKnownPorts_t structure and in particular, DDS_INTEROPERABLE_RTPS_WELL_KNOWN_PORTS.

Most distributed systems can use a single 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 domains. However, even in systems that only use a single 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, *RTI Data Distribution Service* 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 WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9).

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 Ports Used for Discovery (Section 12.5). How *DomainParticipants* discover each other is discussed in Chapter 12.

8.3.5 Setting Up DomainParticipantListeners

DomainParticipants may optionally have *Listeners*. *Listeners* are essentially callback routines and are how *RTI Data Distribution Service* 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 *RTI Data Distribution Service* will propagate the event to the *Entity*'s parent. If the parent *Entity* does not process the event, *RTI Data Distribution Service* 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 Section 8.3.1). You can also set one up after creation time by using the **set_listener()** operation, as illustrated in Figure 8.5 on page 8-21. The **get_listener()** operation can be used to retrieve the current *DomainParticipantListener.*

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 Types of Listeners (Section 4.4.1)).

Note: Due to a thread-safety issue, the destruction of a *DomainParticipantListener* from an enabled *DomainParticipant* should be avoided—even if the *DomainParticipantListener*

Figure 8.5 **Setting up DomainParticipantListener**

```
// 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_PublicationMatchedStatus &status);
        virtual void on_subscription_matched(DDSDataReader *reader,
                        const DDS_SubscriptionMatchedStatus &status);
};
void MyDomainParticipantListener::on_publication_matched(
    DDSDataWriter *writer, const DDS_PublicationMatchedStatus &status)
{
    const char *name = writer->get_topic()->get_name();
    printf("Number of matching DataReaders for Topic %s is %d\n",
            name, status.current_count);
};
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
}
```

8. Domains

has been removed from the *DomainParticipant*. (This limitation does not affect the Java API.)

Note: It is possible for multiple internal *RTI Data Distribution Service* 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:

- ❏ Setting Up TopicListeners (Section 5.1.5)
- ❏ Setting Up PublisherListeners (Section 6.2.5)
- ❏ Setting Up DataWriterListeners (Section 6.3.3)
- ❏ Setting Up SubscriberListeners (Section 7.2.5)
- ❏ Setting Up DataReaderListeners (Section 7.3.3)

8.3.6 Setting DomainParticipant QosPolicies

A *DomainParticipant*'s QosPolicies are used to configure discovery, database sizing, threads, information sent to other *DomainParticipant*s, 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 Restricted Operations in Listener Callbacks (Section 4.5.1).

The DDS_DomainParticipantQos structure has the following format:

```
struct DDS_DomainParticipantQos {
    DDS_UserDataQosPolicy                    user_data;
    DDS_EntityFactoryQosPolicy               entity_factory;
    DDS_WireProtocolQosPolicy                wire_protocol;
    DDS_TransportBuiltinQosPolicy            transport_builtin;
    DDS_TransportUnicastQosPolicy            default_unicast;
    DDS_DiscoveryQosPolicy                   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_TypeSupportQosPolicy                 type_support;
};
```

Table 8.4 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 8.4 **DomainParticipant QosPolicies**

QosPolicy	Description
Database	Various settings and resource limits used by *RTI Data Distribution Service* to control its internal database. See Section 8.5.1.
Discovery	Configures the mechanism used by *RTI Data Distribution Service* to automatically discover and connect with new remote applications. See Section 8.5.2.
DiscoveryConfig	Controls the amount of delay in discovering entities in the system and the amount of discovery traffic in the network. See Section 8.5.3.
DomainParticipantResourceLimits	Various settings that configure how *DomainParticipants* allocate and use physical memory for internal resources, including the maximum sizes of various properties. See Section 8.5.4.
EntityFactory	Controls whether or not child entities are created in the enabled state. See Section 6.4.2.
EntityName	Assigns a name to a *DomainParticipant*. See Section 8.5.5.
Event	Configures the *DomainParticipant's* internal thread that handles timed events. See Section 8.5.6.

Table 8.4 **DomainParticipant QosPolicies**

QosPolicy	Description
Property	Stores name/value(string) pairs that can be used to configure certain parameters of *RTI Data Distribution Service* 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 Section 6.5.15.
ReceiverPool	Configures threads used by *RTI Data Distribution Service* to receive and process data from transports (for example, UDP sockets). See Section 8.5.7.
TransportBuiltin	Specifies which built-in transport plugins are used. See Section 8.5.8.
TransportUnicast	Specifies a subset of transports and port number that can be used by an Entity to receive data. See Section 6.5.21.
TypeSupport	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 Section 6.5.22.
UserData	Along with Topic Data QosPolicy and Group Data QosPolicy, used to attach a buffer of bytes to *RTI Data Distribution Service*'s discovery meta-data. See Section 6.5.23.
WireProtocol	Specifies IDs used by the RTPS wire protocol to create globally unique identifiers. See Section 8.5.9.

8.3.6.1 Configuring QoS Settings when the DomainParticipant is Created

As described in Creating a DomainParticipant (Section 8.3.1), there are different ways to create a *DomainParticipant,* depending on how you want to specify its QoS (with or without a QoS Profile).

- ❏ In Figure 8.4 on page 8-17, we saw an example of how to create a *DomainParticipant* with default QosPolicies 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 Section 8.2.2). Then any *DomainParticipants* created with the *DomainParticipantFactory* will use the new default values. As described in Section 4.1.7, this is a general pattern that applies to the construction of all DDS *Entities*.

- ❑ To create a *DomainParticipant* with non-default QoS without using a QoS Profile, see the example code in Figure 8.6 on page 8-25. 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 8.7 on page 8-26.
- ❑ 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 8.8 on page 8-26.

For more information, see Creating a DomainParticipant (Section 8.3.1) and Chapter 15: Configuring QoS with XML.

Figure 8.6 **Creating a DomainParticipant with Modified QosPolicies (not from a profile)**

```
DDS_DomainId_t domain_id = 10;
DDS_DomainParticipantQos participant_qos;[1]

// 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
DDSDomainParticipant* participant =
                     factory->create_participant(domain_id,
                                                 participant_qos,
                                                 NULL,
                                                 DDS_STATUS_MASK_NONE);
if (participant == NULL) {
    // ... error
}
```

1. Note: In C, you must initialize the QoS structures before they are used, see Section 4.2.2.

8. Domains

Figure 8.7 **Creating a DomainParticipant with a 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 8.8 **Getting QoS Values from a Profile, Changing QoS Values, Creating a DomainParticipant with Modified QoS Values**

```
DDS_DomainParticipantQos participant_qos;
// Get domain participant QoS from profile
retcode = factory->get_participant_qos_from_profile(
                                participant_qos1,
                                "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
}
```

8.3.6.2 Changing QoS Settings After the 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 8.9. It retrieves the current values by calling the *DomainParticipant*'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 *DomainParticipant* has been enabled—this restriction is noted in the descriptions of the individual QosPolicies.
- 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 8.10 on page 8-28. For more information, see Chapter 15: Configuring QoS with XML.

Figure 8.9 **Changing the QoS of an Existing Participant (without a QoS Profile)**

```
DDS_DomainParticipantQos participant_qos;[1]

// 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
}
```

1. For the C API, you need to use DDS_ParticipantQos_INITIALIZER or DDS_ParticipantQos_initialize(). See Special QosPolicy Handling Considerations for C (Section 4.2.2)

8. Domains

Figure 8.10 **Changing the QoS of an Existing Participant with a QoS Profile**

```
retcode = participant->set_qos_with_profile(
                   "ParticipantProfileLibrary","ParticipantProfile");

if (retcode != DDS_RETCODE_OK) {

 // handle error

}
```

8.3.6.3 Getting and Setting the 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 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 Chapter 15: Configuring QoS with XML.

```
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:

```
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 Section 8.3.6.4).

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 Section 8.2.2).

8.3.6.4 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:

```
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:

```
DDS_ReturnCode_t get_default_publisher_qos (DDS_PublisherQos & qos)
```

There are similar operations for *Subscribers*, *DataWriters*, *DataReaders* 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 else, 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.

8.3.7 Looking up Topic Descriptions

The **lookup_topicdescription()** operation allows you to access a locally created **DDSTopicDescription** based on the *Topic's* name.

```
DDSTopicDescription* lookup_topicdescription
                                       (const char *topic_name)
```


8. Domains

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 Section 8.3.7), 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.

8.3.8 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.

```
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 else 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 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 Section 8.3.7).

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 Section 5.1.2).

8.3.9 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*.

1. *Multitopics* are not supported.

There is a similar operation for implicit *Subscribers*:

```
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 Creating Publishers Explicitly vs. Implicitly (Section 6.2.1). 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()**.

The following example code shows how to get the implicit *Publisher/Subscriber.* (For simplicity, error handling is not shown.)

```
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");

publisher->set_qos(publisher_qos);
```

8. Domains

```
/* Get implicit subscriber */
subscriber = participant->get_implicit_subscriber();

/* Change implicit subscriber QoS */
subscriber_qos.partition.name.maximum(3);
subscriber _qos.partition.name.length(3);
subscriber _qos.partition.name[0] = DDS_String_dup("partition_A");
subscriber _qos.partition.name[1] = DDS_String_dup("partition_B");
subscriber _qos.partition.name[2] = DDS_String_dup("partition_C");

subscriber->set_qos(subscriber_qos);
```

8.3.10 Asserting Liveliness

The **assert_liveliness()** operation manually asserts the liveliness of all the *DataWriters* created by this *DomainParticipant* that has LIVELINESS QosPolicy (Section 6.5.11) kind set to **MANUAL_BY_PARTICIPANT**. When **assert_liveliness()** is called, then for those *DataWriters* who have their LIVELINESS set to **MANUAL_BY_PARTICIPANT**, *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 LIVELINESS QosPolicy (Section 6.5.11)

8.3.11 Learning about Discovered DomainParticipants

The **get_discovered_participants()** operation provides you with a list of *DomainParticipants* that have been discovered in the domain (except any that you have said to ignore via the **ignore_participant()** operation (see Section 14.4)).

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 domain and have not been marked as 'ignored.' Otherwise, the operation will fail and return **DDS_RETCODE_PRECONDITION_NOT_MET**. The returned information is of type **DDS_ParticipantBuiltinTopicData**, described in Table 14.1 on page 14-3.

Note: The **get_discovered_participant_data()** operation does not retrieve the **property** information from the builtin-topic data structure. This information is available through the DataReaderListener's **on_data_available()** callback (if a reader listener is installed on the ParticipantBuiltinTopicDataDataReader).

8.3.12 Learning about Discovered Topics

The **get_discovered_topics()** operation provides you with a list of *Topics* that have been discovered in the domain (except any that you have said to ignore via the **ignore_topic()** operation (see Section 14.4)).

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 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 14.4 on page 14-9.

8.3.13 Other DomainParticipant Operations

8.3.13.1 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 Chapter 14: Built-In Topics), various statuses, or from the **get_instance_handle()** operation (see Section 4.1.3).

8.3.13.2 Getting the Current Time

The **get_current_time()** operation returns the current time value from the same time-source (clock) that *RTI Data Distribution Service* uses to timestamp the data published by *DataWriters* (**source_timestamp** of the **SampleInfo** structure, see Section 7.4.5). The time-sources used by *RTI Data Distribution Service* do not have to be synchronized nor are they synchronized by *RTI Data Distribution Service*.

See also: Clock Selection (Section 8.6).

8.3.13.3 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.

8.4 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 8.2 on page 8-7.

- ❏ PROFILE QosPolicy (DDS Extension) (Section 8.4.1)
- ❏ SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension) (Section 8.4.2)

8.4.1 PROFILE QosPolicy (DDS Extension)

This QosPolicy determines the way that XML documents containing QoS profiles are loaded.

All QoS values for DDS Entities can be configured with QoS profiles defined in XML documents. XML documents can be passed to *RTI Data Distribution Service* 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 Chapter 15: Configuring QoS with XML.

This QosPolicy includes the members in Table 8.5 on page 8-35. For the defaults and valid ranges, please refer to the online documentation.

Table 8.5 **DDS_ProfileQosPolicy**

Type	Field Name	Description
DDS_StringSeq	string_profile	Sequence of strings (empty by default) containing a XML document to load. The concatenation of the strings in this sequence must be a valid XML document according to the XML QoS profile schema.
	url_profile	A sequence of URL groups (empty by default) containing a set of XML documents to load. See URL Groups (Section 15.10).
DDS_Boolean	ignore_user_profile	When TRUE, the QoS profiles contained in the file **USER_QOS_PROFILES.xml** in the current working directory will be ignored.
	ignore_environment_profile	When TRUE, the value of the environment variable **NDDS_QOS_PROFILES** will be ignored.
	ignore_resource_profile	When TRUE, the QoS profiles in the file **$NDDSHOME/resource/qos_profiles_4.5*x*[a]/xml/QOS_PROFILES.xml** will be ignored.

a. Replace the x in 4.5x with the version letter for the current *RTI Data Distribution Service* release.

8.4.1.1 Example

```
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);
```

8.4.1.2 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.

8.4.1.3 Related QosPolicies

- None

8.4.1.4 Applicable Entities

- DomainParticipantFactory (Section 8.2)

8.4.1.5 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.

8.4.2 SYSTEM_RESOURCE_LIMITS QoS Policy (DDS Extension)

The SYSTEM_RESOURCE_LIMITS QosPolicy configures *DomainParticipant*-independent resources used by *RTI Data Distribution Service*. 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 8.6. For the default and valid range, please refer to the online documentation.

Table 8.6 **DDS_SystemResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_Long	max_objects_per_thread	Sizes the thread storage that is allocated on a per-thread basis when the thread calls *RTI Data Distribution Service* APIs.

The only parameter that you can set, **max_objects_per_thread**, controls the size of thread-specific storage that is allocated by *RTI Data Distribution Service* for every thread that invokes an *RTI Data Distribution Service* 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 *RTI Data Distribution Service* internal code.

Thus instead of dynamically creating and destroying the objects as a thread enters and leaves different parts of the code, *RTI Data Distribution Service* caches the objects by storing them in thread-specific storage. We assume that a thread will repeatedly call *RTI Data Distribution Service* APIs so that the objects cached will be needed again and again.

The number of objects that will be stored in the cache depends the number of APIs (sections of *RTI Data Distribution Service* 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–independent 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 *RTI Data Distribution Service* 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**.

8.4.2.1 Example

Say an application uses 10 *DomainParticipants*. If a single thread was used to create all 10 *DomainParticipants*, or a single thread is used to call **write()** on *DataWriters* belonging to all 10 participants, it is possible to run out of thread-specific storage. Either the creation of the participant or the **write()** will fail.

In that case, you will need to increase the value of **max_objects_per_thread**.

8.4.2.2 Properties

This QoS policy *cannot* be modified after the *DomainParticipantFactory* is used to create the *first DomainParticipant* in an application.

This QoS can be set differently in different applications.

8.4.2.3 Related QoS Policies

There are no interactions with other QosPolicies.

8.4.2.4 Applicable Entities

- ❏ DomainParticipantFactory (Section 8.2)

8.4.2.5 System Resource Considerations

Increasing the value of **max_objects_per_thread** will increase the amount of memory allocated by *RTI Data Distribution Service* for every thread that access *RTI Data Distribution Service* code. This includes internal *RTI Data Distribution Service* threads as well as user threads. Each object uses about 32 bytes of memory.

8.5 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 8.4 on page 8-23.

- ❏ DATABASE QosPolicy (DDS Extension) (Section 8.5.1)
- ❏ DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2)
- ❏ DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)
- ❏ DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)
- ❏ ENTITYNAME QosPolicy (DDS Extension) (Section 8.5.5)
- ❏ EVENT QosPolicy (DDS Extension) (Section 8.5.6)
- ❏ RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7)
- ❏ TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)
- ❏ WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9)

8.5.1 DATABASE QosPolicy (DDS Extension)

The Database QosPolicy configures how *RTI Data Distribution Service* 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 8.7. For defaults and valid ranges, please refer to the online documentation.

Table 8.7 **DDS_DatabaseQosPolicy**

Type	Field Name	Description
DDS_ThreadSettings_t	thread.mask thread.priority thread.stack_size	Thread settings for the database thread used by *RTI Data Distribution Service* to periodically remove deleted records from the database. The values used for these settings are OS-dependent. Note: thread.cpu_list and thread.cpu_rotation are not relevant in this QoS policy.
DDS_Duration_t	shutdown_timeout	The maximum time that the *DomainParticipant* will wait for the database thread to terminate when the participant is destroyed.
DDS_Duration_t	cleanup_period	The period at which the database thread wakes up to removed deleted records.
DDS_Duration_t	shutdown_cleanup_period	The period at which the database thread wakes up to removed deleted records when the *DomainParticipant* is being destroyed.
DDS_Long	initial_records	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.
DDS_Long	max_skiplist_level	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 log2(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 domain, which ever is greater, can be used to set this parameter.

Table 8.7 **DDS_DatabaseQosPolicy**

Type	Field Name	Description
DDS_Long	max_weak_ references	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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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.
DDS_Long	initial_weak_ references	The initial number of entries in the weak reference table. See max_weak_references. *RTI Data Distribution Service* 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.

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 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4) 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 *RTI Data Distribution Service* relative to each other as well as relative to non-*RTI Data Distribution Service* threads in their applications. Chapter 17: RTI Data Distribution Service Threading Model, EVENT QosPolicy (DDS Extension) (Section 8.5.6), and RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7) discuss the other threads that are created by *RTI Data Distribution Service*.

A record in the database can be deleted only when no threads are using it. *RTI Data Distribution Service* 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.

8.5.1.1 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.

8.5.1.2 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.1.3 Related QosPolicies

- ❏ DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)
- ❏ EVENT QosPolicy (DDS Extension) (Section 8.5.6)
- ❏ RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7)

8.5.1.4 Applicable Entities

- ❏ DomainParticipants (Section 8.3)

8.5.1.5 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.

RTI Data Distribution Service 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.

8.5.2 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 Chapter 12: Discovery.

This QosPolicy includes the members in Table 8.8. For defaults and valid ranges, please refer to the online documentation.

Table 8.8 **DDS_DiscoveryQosPolicy**

Type	Field Name	Description
DDS_StringSeq	enabled_transports	Transports available for use by the discovery process. See Section 8.5.2.1.
DDS_StringSeq	initial_peers	Unicast locators (address/indices) of potential participants with which this *DomainParticipant* will attempt to establish communications. See Section 8.5.2.2.
DDS_StringSeq	multicast_receive_ addresses	List of multicast addresses on which Discovery-related messages can be received by the *DomainParticipant*. See Section 8.5.2.4.
DDS_Long	metatraffic_transport _priority	Transport priority to be used for sending Discovery messages. See Section 8.5.2.5.
DDS_Boolean	accept_unknown_ peers	Whether to accept a participant discovered via unicast that is not in the **initial_peers** list. See Section 8.5.2.6.

8.5.2.1 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 online documentation (**Modules, *RTI Data Distribution Service* API Reference, Pluggable Transports**) for more information.

8.5.2.2 Setting the 'Initial Peers' List

When a *DomainParticipant* is created, it needs to find other participants in the same domain—this is known as the 'discovery process' which is discussed in Chapter 12: Discovery. 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 Peer Descriptor Format (Section 12.2.1).

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 *RTI Data Distribution Service* discovery process will try to contact all potential peer participants in the list periodically using unicast transports (as configured by the DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)).

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 Configuring the Peers List Used in Discovery (Section 12.2).

Note: IPv4 multicast addresses must have a prefix

When using the UDPv6 transport: if there are any IPv4 multicast addresses in the peers list, make sure they have "**udpv4://**" in front of them.

For example:

```
setenv NDDS_DISCOVERY_PEERS
        "udpv4://localhost,udpv4://239.255.0.1,shmem://"
```

or, to add IPv6 loopback and an IPv6 multicast address:

```
setenv NDDS_DISCOVERY_PEERS
        "udpv4://localhost,udpv4://239.255.0.1,
        shmem://,udpv6://::1,udpv6://ff05::239.255.0.1"
```

8.5.2.3 Adding to the Peers List

The *DomainParticipant*'s **add_peer()** operation adds a peer description to the internal peer list that was initialized by the **initial_peer** field of the DISCOVERY QosPolicy.

```
DDS_ReturnCode_t DDSDomainParticipant::add_peer
                                        (const char* peer_desc)
```

The **peer_desc** string must be formatted as specified in Peer Descriptor Format (Section 12.2.1).

You can call this operation any time after the *DomainParticipant* has been created (regardless of whether or not it is enabled). If the *DomainParticipant* is already 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** (Section 8.5.2.6) set to FALSE and still use **add_peer()**.)

Note: there is no complementary "remove peer" operation. You can, however, ignore data from a participant by using the **ignore_participant()** operation described in Section 14.4.

8.5.2.4 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 Configuring the Peers List Used in Discovery (Section 12.2). 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, *RTI Data Distribution Service* 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.

8.5.2.5 Meta-Traffic Transport Priority

The **metatraffic_transport_priority** field is used to specify the transport priority to be used for sending all discovery meta-traffic. See the TRANSPORT_PRIORITY QosPolicy (Section 6.5.19) for details on how transport priorities may be used.

Currently, the builtin transports provided by *RTI Data Distribution Service* will ignore the value set in this field.

8.5.2.6 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 Section 8.5.2.3).

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:

- ❑ 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**.

8.5.2.7 Example

You will always use this policy to set the **participant_id** when you want to run more than one *DomainParticipant* in the same 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.

8.5.2.8 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.2.9 Related QosPolicies

❏ DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)

❑ TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)

8.5.2.10 Applicable Entities

❑ DomainParticipants (Section 8.3)

8.5.2.11 System Resource Considerations

For every entry in the **initial_peers** list, *RTI Data Distribution Service* 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.

8.5.3 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 (e.g. 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 8.9. Many of these members are described in Chapter 12: Discovery. For defaults and valid ranges, please refer to the online documentation.

Table 8.9 **DDS_DiscoveryConfigQosPolicy**

Type	Field Name	Description
DDS_Duration_t	participant_liveliness_ lease_duration	The time period after which other *DomainParticipants* can consider this one dead if they do not receive a liveliness packet from this *DomainParticipant*.
DDS_Duration_t	participant_liveliness_ assert_period	The period of time at which this *DomainParticipant* will send out packets asserting that it is alive.

Table 8.9 **DDS_DiscoveryConfigQosPolicy**

Type	Field Name	Description
DDS_RemoteParticipantPurgeKind	remote_participant_purge_kind	Controls the *DomainParticipant*'s behavior for purging records of remote participants (and their contained entities) with which discovery communication has been lost. See Section 8.5.3.2.
DDS_Duration_t	max_liveliness_loss_detection_period	The maximum amount of time between when a remote entity stops maintaining its liveliness and when the matched local entity realizes that fact.
DDS_Long	initial_participant_announcements	Sets how many initial liveliness announcements the *DomainParticipant* will send when it is first enabled, or after discovering a new remote participant.
DDS_Duration_t	min_initial_participant_announcement_period	Sets the minimum and maximum times between liveliness announcements.
DDS_Duration_t	max_initial_participant_announcement_period	When a participant is first enabled, or after discovering a new remote participant, *RTI Data Distribution Service* sends initial_paricipant_annoucements 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.
DDS_BuiltinTopicReaderResourceLimits_t	participant_reader_resource_limits	Configures the resource for the built-in *DataReaders* used to access discovery information; see Section 8.5.3.1 and Chapter 14: Built-In Topics.
	publication_reader_resource_limits	
	subscription_reader_resource_limits	

Table 8.9 **DDS_DiscoveryConfigQosPolicy**

Type	Field Name	Description
DDS_RtpsReliableWriter Protocol_t	publication_writer subscription_writer	Configures the RTPS reliable protocol parameters for the writer side of a reliable connection. These *DataWriters* send reliable discovery information to *DomainParticipants* that were dynamically discovered (see Chapter 12: Discovery). DDS_RtpsReliableWriterProtocol_t is described in Table 6.29 on page 6-95.
DDS_RtpsReliableReader Protocol_t	publication_reader subscription_reader	Configures the RTPS reliable protocol parameters for the reader side of a reliable connection. These *DataReaders* receive discovery information reliably from *DomainParticipants* that were dynamically discovered (see Chapter 12: Discovery). DDS_RtpsReliableReaderProtocol_t is described in Table 7.19 on page 7-76.
DDS_DiscoveryConfig BuiltinPluginKindMask	builtin_discovery_ plugins	The kind mask for selecting built-in discovery plugins: ❑ Simple Discovery Protocol: DDS_DISCOVERYCONFIG_ BUILTIN_SDP ❑ Enterprise Discovery Service: DDS_DISCOVERYCONFIG_ BUILTIN_EDS (Requires a separately purchased product, *RTI Enterprise Discovery Service*.)

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 DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2). 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

participant_liveliness_assert_period < 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 liveliness messages are sent, the *DomainParticipant* will return to sending liveliness 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 Chapter 12: Discovery.

8.5.3.1 Resource Limits for Builtin-Topic DataReaders

The **DDS_BuiltinTopicReaderResourceLimits_t** structure is shown in Table 8.10. 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*.

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 Chapter 14: Built-In Topics 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*.

Table 8.10 **DDS_BuiltinTopicReaderResourceLimits_t**

Type	Field Name	Description
DDS_Long	initial_samples	Initial number of meta-traffic data samples that can be stored by a builtin-topic *DataReader*.
	max_samples	Maximum number of meta-*traffic* data samples that can be stored by a builtin-topic DataReader.
	initial_infos	Initial number of DDS_SampleInfo structures allocated for the builtin-topic *DataReader*.
	max_infos	Maximum number of DDS_SampleInfo structures that can be allocated for the built-in topic *DataReader*. *max_infos* must be >= *max_samples*
	initial_outstanding_reads	Initial number of times in which memory can be concurrently loaned via read/take calls on the builtin-topic *DataReader* without being returned with return_loan().
	max_outstanding_reads	Maximum number of times in which memory can be concurrently loaned via read/take calls on the builtin-topic *DataReader* without being returned with return_loan().

8.5.3.2 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 have 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 *RTI Data Distribution Service* 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 DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)).

A participant will maintain its own liveliness to any remote participant via inter-participant liveliness traffic (see LIVELINESS QosPolicy (Section 6.5.11)).

The default Simple Discovery Protocol described in Chapter 12: Discovery automatically maintains this liveliness, whereas other discovery mechanisms may or may not.

❑ **DDS_NO_REMOTE_PARTICIPANT_PURGE**

With this value, *RTI Data Distribution Service* 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, but may occur if you are using *RTI Enterprise Discovery Service*. [1]

❑ 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 LIVELINESS QosPolicy (Section 6.5.11) **kind**.

8.5.3.3 Controlling the Reliable Protocol Used by Builtin-Topic DataWriters/DataReaders

The connection between the *DataWriters* and *DataReaders* for the publication and subscription builtin-topics are reliable. The **publication_writer**, **subscription_writer**, **publication_reader**, and **subscription_reader** parameters of the DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3) configure the reliable messaging protocol

1. *RTI Enterprise Discovery Service* is an optional package that provides participant-matching services for *RTI Data Distribution Service* applications.

used by *RTI Data Distribution Service* for those topics. *RTI Data Distribution Service*'s reliable messaging protocol is discussed in Chapter 10: Reliable Communications.

See also:

- DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)
- DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1).

8.5.3.4 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.

8.5.3.5 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.3.6 Related QosPolicies

- DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2)
- DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4)
- WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9)
- DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2)
- DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1)
- DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2)

8.5.3.7 Applicable Entities

- DomainParticipants (Section 8.3)

8.5.3.8 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 *RTI Data Distribution Service* may allocate for a *DomainParticipant* while increasing the initial values will increase the initial memory allocated for a *DomainParticipant*.

8.5.4 DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension)

The DOMAIN_PARTICIPANT_RESOURCE_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 DDS 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 DDS 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 prevent *DomainParticipants* from allocating memory after the initialization period.

The maximum sizes of several variable-length parameters—such as the number of partitions that can be stored in the PARTITION QosPolicy (Section 6.4.5), the maximum length of data store in the USER_DATA QosPolicy (Section 6.5.23) and GROUP_DATA QosPolicy (Section 6.4.4), and many others—can be changed from their defaults using this QoS. However, it is important that <u>all *DomainParticipants* that need to communicate with each other use the same set of maximum values.</u> 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 8.11. For defaults and valid ranges, please refer to the online documentation.

Table 8.11 **DDS_DomainParticipantResourceLimitsQosPolicy**

Type	Field Name	Description

DDS_Allocation-Settings_t (see description column)	local_writer_allocation
	local_reader_allocation
	local_publisher_allocation
	local_subscriber_allocation
	local_topic_allocation
	remote_writer_allocation
	remote_reader_allocation
	remote_participant_allocation
	matching_writer_reader_pair_allocation
	matching_reader_writer_pair_allocation
	ignored_entity_allocation
	content_filtered_topic_allocation
	content_filter_allocation
	read_condition_allocation
	query_condition_allocation
	outstanding_asynchronous_sample_allocation
	flow_controller_allocation
		Each allocation structure configures how many objects of each type, <object>_allocation, will be allocated by the *DomainParticipant*.
		See Configuring Resource Limits for Asynchronous DataWriters (Section 8.5.4.1).
		DDS_AllocationSettings_t
		{
		DDS_Long initial_count;
		DDS_Long max_count;
		DDS_Long incremental_count;
		};

DDS_Long	local_writer_hash_buckets local_reader_hash_buckets local_publisher_hash_buckets local_subscriber_hash_buckets local_topic_hash_buckets remote_writer_hash_buckets remote_reader_hash_buckets remote_participant_hash_buckets matching_writer_reader_pair_hash_buckets matching_reader_writer_pair_hash_buckets ignored_entity_hash_buckets content_filtered_topic_hash_buckets content_filter_hash_buckets flow_controller_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.

Table 8.11 **DDS_DomainParticipantResourceLimitsQosPolicy**

<table>
<tr><th>Type</th><th>Field Name</th><th>Description</th></tr>
<tr><td>DDS_Long</td><td>max_gather_destinations</td><td>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.</td></tr>
<tr><td rowspan="6">DDS_Long</td><td>participant_user_data_max_length</td><td rowspan="6">Controls the maximum lengths of USER_DATA QosPolicy (Section 6.5.23), TOPIC_DATA QosPolicy (Section 5.2.1) and GROUP_DATA QosPolicy (Section 6.4.4) for different entities.
Must be configured to be the same values on all DomainParticipants in the same domain.</td></tr>
<tr><td>topic_data_max_length</td></tr>
<tr><td>publisher_group_data_max_length</td></tr>
<tr><td>subscriber_group_data_max_length</td></tr>
<tr><td>writer_user_data_max_length</td></tr>
<tr><td>reader_user_data_max_length</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 PARTITION QosPolicy (Section 6.4.5).
Must be configured to be the same value on all DomainParticipants in the same 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 PARTITION QosPolicy (Section 6.4.5).
Must be configured to be the same value on all DomainParticipants in the same 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 Using Generated Types without RTI Data Distribution Service (Standalone) (Section 3.7).</td></tr>
<tr><td>DDS_Long</td><td>contentfilter_property_max_length</td><td>Maximum length of all data related to ContentFilteredTopics (Section 5.4).</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 MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12).</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 MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12).</td></tr>
</table>

Table 8.11 **DDS_DomainParticipantResourceLimitsQosPolicy**

Type	Field Name	Description
DDS_Long	participant_property_list_max_length	Maximum number of properties ((name, value) pairs) that can be stored in the *DomainParticipant's* PROPERTY QosPolicy (DDS Extension) (Section 6.5.15).
	participant_property_string_max_length	Maximum cumulative length (in bytes) of all the (name, value) pairs in a *DomainParticipant's* Property QosPolicy.
	writer_property_list_max_length	Maximum number of properties ((name, value) pairs) that can be stored in a *DataWriter's* Property QosPolicy.
	writer_property_string_max_length	Maximum cumulative length (in bytes) of all the (name, value) pairs in a *DataWriter's* Property QosPolicy.
	reader_property_list_max_length	Maximum number of properties ((name, value) pairs) that can be stored in a *DataReader's* Property QosPolicy.
	reader_property_string_max_length	Maximum cumulative length (in bytes) of all the (name, value) pairs in a *DataReader's* Property QosPolicy.

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_code_max_serialized_length**. This parameter limits the size of the type code that a *DomainParticipant* is able to store and propagate for user data types. Type codes can be used by external applications to understand user data types without having the data type predefined in 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 type codes larger than the default maximum of 2048 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_code_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 efficient.

If you modify any of the **<entity type>_data_max_length**, **max_partitions**, or **max_partition_cummulative_characters** parameters, then you must make sure that they are modified to be the same value for all *DomainParticipants* in the same 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.

8.5.4.1 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 RESOURCE_LIMITS QosPolicy (Section 6.5.18)).

8.5.4.2 Configuring Memory Allocation

The **<object type>_allocation** configure the number of <object type>'s that can be stored in the internal *RTI Data Distribution Service* 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. Memory is allocated for the storage of the objects, thus **initial_count** will determine how much memory is initially allocated, and **max_count** will determine the maximum amount of memory that *RTI Data Distribution Service* is allowed to allocate. The **incremental_count** is used to allocate more memory in chunks when the number of objects created exceed the **initial_count**.

You should modify these parameters only if you want to decrease the initial memory used by *RTI Data Distribution Service* when a *DomainParticipant* is created or increase the maximum number of local and remote *Entities* that can be stored in a *DomainParticipant*.

How *RTI Data Distribution Service* is allowed to allocate memory for a *DomainParticipant* after initialization depends on how you set these parameters.

1. Static memory allocation

   No memory is allocated by *RTI Data Distribution Service* after creation. Set **initial_count = max_count**. The **incremental_count** should be set to 0.

   Advantage: All memory allocation is done when creating the *DomainParticipant*; no dynamic allocation during run-time. You know immediately if you have enough memory to run in that configuration.

   Disadvantage: Requires a fairly static system and/or good estimates on the number of *Entities* in the distributed system. *RTI Data Distribution Service* will fail to execute properly once the number of *Entities* exceed the configure bounds.

2. Dynamic, bounded allocation

   Set **initial_count** to configure the initial amount of memory to be allocated. Set *max_count* to the maximum allowable upper bound (see the online documentation).

   Advantage: Initial memory usage may be lower and memory is allocated as needed and only if needed.

   Disadvantage: *RTI Data Distribution Service* may allocate memory dynamically which may have an impact on performance.

   If you allow *RTI Data Distribution Service* to allocate memory dynamically, you can either:

   - Use fixed-size increments (set **incremental_count** to the desired fixed size).

     Advantage: well known amount of memory allocated each time.

     Disadvantage: may require more frequent allocations.

   - Double the amount of extra memory allocated each time memory is needed (set **incremental_count** to -1).

     Advantage: requires fewer allocations.

     Disadvantage: may allocate considerably more memory than is really needed.

8.5.4.3 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 domain use the same sets of values or else it is possible that *RTI Data Distribution Service* will not successfully connect two *Entities*.

8.5.4.4 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.4.5 Related QosPolicies

- DATABASE QosPolicy (DDS Extension) (Section 8.5.1)
- DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)
- MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12)
- USER_DATA QosPolicy (Section 6.5.23)
- TOPIC_DATA QosPolicy (Section 5.2.1)
- GROUP_DATA QosPolicy (Section 6.4.4)
- PARTITION QosPolicy (Section 6.4.5)
- PROPERTY QosPolicy (DDS Extension) (Section 6.5.15)

8.5.4.6 Applicable Entities

- DomainParticipants (Section 8.3)

8.5.4.7 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.

8.5.5 ENTITYNAME QosPolicy (DDS Extension)

The ENTITYNAME QosPolicy is used to assign a string name to a DomainParticipant. How this name is used is strictly application-dependent.

It is useful to attach names to *DomainParticipants* that are meaningful to the user. These names are propagated during discovery so that applications can use these names to identify, in a user-context, the *DomainParticipants* that it discovers. Also, RTI tools such as *RTI Analyzer* will print these names for *DomainParticipants* that it finds so that users can easily determine exactly which application they are browsing with *RTI Analyzer*.

This name will appear in the builtin topic for the *DomainParticipant*, as seen in Table 14.1 on page 14-3.

It contains the member listed in Table 8.13.

Table 8.12 **DDS_EntityNameQoSPolicy**

Type	Field Name	Description
char *	name	A null terminated string, up to 255 characters in length.

Prior to **get_qos()**, if the **name** field in this QosPolicy is not null, *RTI Data Distribution Service* assumes the memory to be valid and big enough and may write to it. If that is not desired, set **name** to NULL before calling **get_qos()** and *RTI Data Distribution Service* will allocate adequate memory for name.

When the destructor of DomainParticipantQos (in C++, C++/CLI, and C#) or **DomainParticipantQos_finalize()** (in C) is called, *RTI Data Distribution Service* will attempt to free the memory used for **name** if it is not NULL. If this behavior is not desired, set **name** to NULL before DomainParticipantQos destructor is called or before **DomainParticipantQos_finalize()** is called.

8.5.5.1 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is enabled.

8.5.5.2 Related QosPolicies

- None

8.5.5.3 Applicable Entities

- DomainParticipants (Section 8.3)

8.5.5.4 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.

8.5.6 EVENT QosPolicy (DDS Extension)

The EVENT QosPolicy configures the internal *RTI Data Distribution Service* 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 8.13. For defaults and valid ranges, please refer to the online documentation.

Table 8.13 **DDS_EventQoSPolicy**

Type	Field Name	Description
DDS_Thread Settings_t	thread.mask thread.priority thread.stack_size	Thread settings for the event thread used by *RTI Data Distribution Service* to wake up for a timed event and possibly execute listener callbacks. The values used for these settings are OS-dependent. Note: thread.cpu_list and thread.cpu_rotation are not relevant in this QoS policy.
DDS_Long	initial_count	Initial number of events that can be stored simultaneously.
DDS_Long	max_count	Maximum number of events that can be stored simultaneously.

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 Event Thread (Section 17.2).

For example, a *DataReader* may have a constraint that requires data to be received within a period of time specified by the DEADLINE QosPolicy (Section 6.5.4). 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 Chapter 10: Reliable Communications.

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.

8.5.6.1 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 deadlines 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.

8.5.6.2 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.6.3 Related QosPolicies

- DATABASE QosPolicy (DDS Extension) (Section 8.5.1)
- RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7)

8.5.6.4 Applicable Entities

- DomainParticipants (Section 8.3)

8.5.6.5 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 an 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.

8.5.7 RECEIVER_POOL QosPolicy (DDS Extension)

The RECEIVER_POOL QosPolicy configures the internal *RTI Data Distribution Service* thread used to process the data received from a transport. The Receive thread is described in detail in Section 17.3.

This QosPolicy contains the members in Table 8.14. For defaults and valid ranges, please refer to the online documentation.

Table 8.14 **DDS_ReceiverPoolQoSPolicy**

Type	Field Name	Description
struct DDS_ThreadSettings_t	thread.mask thread.priority thread.stack_size hread.cpu_list thread.cpu_rotation	Thread settings for the receive thread(s) used by *RTI Data Distribution Service* to process data received from a transport. The values used for these settings are OS-dependent. See also: Controlling CPU Core Affinity for RTI Threads (Section 17.5).
DDS_Long	buffer_size	Size of the receive buffer in bytes.
DDS_Long	buffer_alignment	Byte-alignment of the receive buffer.

This QosPolicy sets the thread properties, like priority level and stack size, for the threads used to receive and process data from transports. *RTI Data Distribution Service* uses a separate receive thread per port per transport plugin. To force *RTI Data Distribution Service* 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) (Section 6.5.21) or TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5) for the *DataReader*.

RTI Data Distribution Service 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 samples received for the participant's *DataReaders*, as well as messages used by *RTI Data Distribution Service* itself in support of the application discovery process discussed in Chapter 12.

The user application may configure *RTI Data Distribution Service* to create many more threads for receiving data sent via multicast or even to dedicate a thread to process the data samples of a single *DataReader* received on a particular transport. This QosPolicy is used in the creation of all receive threads.

In many applications, users change the configuration of the builtin-transport **message_size_max** property to increase the size of the largest data packet that can be sent or received through the transport. Typically, users change the UDPv4 transport plugin's **message_size_max** to 65536 (64 K), which is the largest packet that can be sent/

received via UDP. The ReceiverPool QosPolicy's **buffer_size** should be set to at least the same value as the maximum **message_size_max** parameter across all of the transports being used that does not support zero-copy. (A transport that supports zero-copy will not use the receive buffer. The only built-in transport that supports zero-copy is the UDPv4 transport on VxWorks platforms.) If you are using the default configuration of the built-in transports, you should not need to change this buffer size.

In addition, if your application *only* uses transports that support zero-copy, then you do not need to modify the value of **buffer_size**, even if the **message_size_max** of the transport is changed. Transports that support zero-copy do not copy their data into the buffer provided by the receive thread. Instead, they provide the receive thread data in a buffer allocated by the transport itself. The only built-in transport that supports zero-copy is the UDPv4 transport on VxWorks platforms.

8.5.7.1 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 respect to the other threads in the application as appropriate for the proper operation of the system.

8.5.7.2 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.7.3 Related QosPolicies

- DATABASE QosPolicy (DDS Extension) (Section 8.5.1)
- EVENT QosPolicy (DDS Extension) (Section 8.5.6)

8.5.7.4 Applicable Entities

- DomainParticipants (Section 8.3)

8.5.7.5 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.

8.5.8 TRANSPORT_BUILTIN QosPolicy (DDS Extension)

RTI Data Distribution Service comes with three different transport plugins built into the core libraries (for most supported target platforms). These are plugins for UDPv4, shared memory, and UDPv6.

This QosPolicy allows you to control which built-in 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.

It contains the member in Table 8.15. For the default and valid values, please refer to the online documentation.

Table 8.15 **DDS_TransportBuiltinQosPolicy**

Type	Field Name	Description
DDS_TransportBuiltinKindMask	mask	A mask with bits that indicate which built-in transports will be installed.

Please see the online documentation (**Modules, *RTI Data Distribution Service* API Reference, Pluggable Transports, Using Transport Plugins and Built-in Transport Plugins**) for more information.

See also: “Note:” on page 8-43.

8.5.8.1 Example

See Section 8.5.8.5 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.

8.5.8.2 Properties

This QosPolicy cannot be modified after the *DomainParticipant* is created.

It can be set differently on the publishing and subscribing sides.

8.5.8.3 Related QosPolicies

- ❏ TRANSPORT_SELECTION QosPolicy (DDS Extension) (Section 6.5.20)

- ❑ TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21)
- ❑ TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5)

8.5.8.4 Applicable Entities

- ❑ DomainParticipants (Section 8.3)

8.5.8.5 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.

8.5.9 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 *RTI Data Distribution Service* to format and interpret messages between *DomainParticipants*.

It includes the members in Table 8.16. For defaults and valid ranges, please refer to the online 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 8.16 **DDS_WireProtocolQosPolicy**

Type	Field Name	Description
DDS_Long	participant_id	Unique identifier for participants that belong to the same domain on the same host. See Section 8.5.9.1.

Table 8.16 **DDS_WireProtocolQosPolicy**

Type	Field Name	Description
DDS_UnsignedLong	rtps_host_id	A machine/OS-specific host ID, unique in the domain. See Section 8.5.9.2.
	rtps_app_id	A participant-specific ID, unique within the scope of the **rtps_host_id**. See Section 8.5.9.2.
	rtps_instance_id	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 Section 8.5.9.2.
DDS_RtpsWellKnownPorts_t	rtps_well_known_ports	Determines the well-known multicast and unicast ports for discovery and user traffic. See Section 8.5.9.3.
DDS_RtpsReservedPortKindMask	rtps_reserved_ports_mask	Specifies which well-known multicast and unicast ports to reserve when enabling the *DomainParticipant*.
DDS_WireProtocolQosPolicyAutoKind	rtps_auto_id_kind	Kind of auto mechanism used to calculate the GUID prefix.

Note that DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2) and DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1) configure RTPS and reliability properties on a per *DataWriter* and *DataReader* basis.

8.5.9.1 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 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 domain on the same host. You can use the same **participant_id** value for *DomainParticipants* in the same 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 Ports Used for Discovery (Section 12.5). If you only have one *DomainParticipant* in the same domain on the same host, you will not need to modify this value.

You can either allow *RTI Data Distribution Service* 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 *RTI Data Distribution Service* will select a participant ID for you.

RTI Data Distribution Service 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 Inbound Ports for Meta-Traffic (Section 12.5.1) an Inbound Ports for User Traffic (Section 12.5.2). By default, *RTI Data Distribution Service* will reserve the meta-traffic unicast port, the meta-traffic multicast port, and the user traffic unicast port.

RTI Data Distribution Service 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 *RTI Data Distribution Service* has selected, either:

- Change the verbosity level of the NDDS_CONFIG_LOG_CATEGORY_API category to NDDS_CONFIG_LOG_VERBOSITY_STATUS_LOCAL (see Controlling Messages from RTI Data Distribution Service (Section 18.2)).
- Call **get_qos()** and look at the **participant_id** value in the WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9) 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 Configuring the Peers List Used in Discovery (Section 12.2) 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 Ports Used for Discovery (Section 12.5)), and for IP, a port number cannot be larger than 65536.

For details, see Ports Used for Discovery (Section 12.5).

8.5.9.2 Host, App, and Instance IDs

The **rtps_host_id**, **rtps_app_id**, and **rtps_instance_id** values are used by the RTPS protocol to allow *RTI Data Distribution Service* to distinguish messages received from different *DomainParticipants*. Their combined values must be globally unique across all existing *DomainParticipants* in the same domain. In addition, if an application dies unexpectedly and 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.

If the value of **rtps_host_id** is set to **DDS_RTPS_AUTO_ID**, the IPv4 address of the host is used as the host ID. If the host does not have an IPv4 address, then you should set this value to uniquely distinguish the host from other nodes in the system.

If the value of **rtps_app_id** is set to **DDS_RTPS_AUTO_ID**, the process (or task) ID is used. There can be at most 256 distinct participants in a shared address space (process) with a unique **rtps_app_id**.

If the value of **rtps_instance_id** is set to **DDS_RTPS_AUTO_ID**, a counter is assigned that is incremented per new participant. Thus, together with **rtps_app_id**, there can be at most 2^64 distinct participants in a shared address space with a unique RTPS Globally Unique Identifier (GUID).

8.5.9.3 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 *RTI Data Distribution Service*) and user traffic (from your application).

It includes the members in Table 8.17. For defaults and valid ranges, please refer to the online documentation.

Table 8.17 **DDS_RtpsWellKnownPorts_t**

Type	Field Name	Description
DDS_Long	port_base	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.
	domain_id_gain	Tunable gain parameters. See Ports Used for Discovery (Section 12.5).
	participant_id_gain	
	builtin_multicast_port_offset	Additional offset for meta-traffic port. See Inbound Ports for Meta-Traffic (Section 12.5.1).
	builtin_unicast_port_offset	
	user_multicast_port_offset	Additional offset for user traffic port. See Inbound Ports for User Traffic (Section 12.5.2).
	user_unicast_port_offset	

8.5.9.4 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 domain. In order to make it easier to control how this 96-bit value is generated, *RTI Data Distribution Service* divides it into three integers: a host ID, the value of which is based on the identity of the machine on which the participant is executing, an application ID (whose value is based on the process or task in which the participant is contained), and an instance ID which identifies the participant itself.

This QoS policy provides you with a choice of algorithms for generating these values automatically. In case none of these algorithms suit your needs, you may also choose to specify some or all of them yourself.

The following three fields compose the GUID prefix and by default are set to DDS_RTPS_AUTO_ID. The meaning of this flag depends on the value assigned to **rtps_auto_id_kind**.

- ❑ **rtps_host_id**
- ❑ **rtps_app_id**
- ❑ **rtps_instance_id**

Depending on the **rtps_auto_id_kind** value, there are two different scenarios:

Scenario 1:

In the default and most common scenario, **rtps_auto_id_kind** is set to DDS_RTPS_AUTO_ID_FROM_IP. Doing so, each field is interpreted 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
- ❑ **rtps_app_id**: the process (or task) ID is assigned
- ❑ **rtps_instance_id**: A counter is assigned that is incremented per new participant

Note: If the IP address assigned to the interface is not unique within the network (for instance, if it is not configured), then is it possible that the GUID (specifically, the **rtps_host_id** portion) may also not be unique.

Scenario 2:

In this situation, *RTI Data Distribution Service* provides a different value for **rtps_auto_id_kind**: DDS_RTPS_AUTO_ID_FROM_MAC. As the name suggests, this alternative mechanism uses the MAC address instead of the IPv4 address. Since the MAC address size is up to 64 bits, the logical mapping of the host information, the application ID, and the instance identifiers has to change.

Note to Solaris Users: To use DDS_RTPS_AUTO_ID_FROM_MAC, you must run the DDS application while logged in as 'root.'

Using DDS_RTPS_AUTO_ID_FROM_MAC, the default value of each field is interpreted 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.

If DDS_RTPS_AUTO_ID_FROM_MAC is used, the **rtps_instance_id** has been logically split into two parts: 24 bits for the process/task ID and 8 bits for the per new participant counter. To give to users the ability to manually set the two parts independently, a bit field mechanism has been introduced for the **rtps_instance_id** field when it is used in combination with DDS_RTPS_AUTO_ID_FROM_MAC. If one of the two parts is set to 0, only this part will be handled by RTI Data Distribution Service and you will be able to handle the other one manually.

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.

The first step is to get the *DomainParticipant* QoS from the DomainParticipantFactory:

```
DDS_DomainParticipantFactory_get_default_participant_qos(
           DDS_DomainParticipantFactory_get_instance(),
           &participant_qos);
```

Next, change the WireProtocolQosPolicy using one of the following options.

Then create the *DomainParticipant* as usual using the modified QoS structure instead of the default one.

Option 1. Use DDS_RTPS_AUTO_ID_FROM_MAC to explicitly set just the application/task identifier portion of the **rtps_instance_id** field:

```
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));
```

Option 2. Only set the per participant counter and let *RTI Data Distribution Service* handle the application/task identifier:

```
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));
```

Option 3. Set the entire **rtps_instance_id** field yourself:

```
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 *RTI Data Distribution Service* will take responsibility for them).

RTI recommends that you always specify the two parts separately in order to avoid errors.

Option 4. Let *RTI Data Distribution Service* handle the entire **rtps_instance_id** field:

```
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 *RTI Data Distribution Service* will take responsibility for them).

RTI recommends that you always specify the two parts separately in order to clearly show the difference.

8.5.9.5 Example

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 *RTI Data Distribution Service*'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 or LynxOS, 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 is used 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 DISCOVERY_CONFIG QosPolicy 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.

8.5.9.6 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 domain across all applications. The value of **rtps_app_id** should also between different invocations of the same application; for example, when an application is restarted.

8.5.9.7 Related QosPolicies

- DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)

8.5.9.8 Applicable Entities

- DomainParticipants (Section 8.3)

8.5.9.9 System Resource Considerations

The use of this policy does not significantly impact the use of resources.

8.6 Clock Selection

RTI Data Distribution Service 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 Section 8.3.13.2).

8.6.1 Available Clocks

Two clock implementations are generally available: the *realtime* clock and the *monotonic* clock.

The realtime 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**".

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 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**".

8.6.2 Clock Selection Strategy

To configure the clock selection, use the *DomainParticipant's* PROPERTY QosPolicy (DDS Extension) (Section 6.5.15). Table 8.18 lists the supported properties.

Table 8.18 **Clock Selection Properties**

Property	Description
dds.clock.external_clock	Comma-delimited list of clocks to use for the external clock, in the order of preference. Valid clock names are "realtime", "system", or "monotonic".
dds.clock.internal_clock	Comma-delimited list of clocks to use for the internal clock, in the order of preference. Valid clock names are "realtime", "system", or "monotonic".

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 8.18 on page 8-76). 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.

Chapter 9 Building Applications

This chapter provides instructions on how to build *RTI Data Distribution Service* applications for the following platforms:

- ❑ UNIX-based Platforms (Section 9.3) (including Solaris™, Red Hat® and Yellow Dog™ Linux, QNX®, and LynxOS® systems)
- ❑ Windows Platforms (Section 9.4)
- ❑ Java Platforms (Section 9.5)

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 Platform Notes.

To build a non-Java application using *RTI Data Distribution Service*, you must specify the following items:

- ❑ **NDDSHOME** environment variable
- ❑ *RTI Data Distribution Service* header files
- ❑ *RTI Data Distribution Service* libraries to link
- ❑ Compatible system libraries
- ❑ Compiler options

To build Java applications using *RTI Data Distribution Service*, you must specify the following items:

- ❑ **NDDSHOME** environment variable
- ❑ *RTI Data Distribution Service* JAR file
- ❑ 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 Platform Notes.

9.1 Running on a Computer Not Connected to a Network

If you want to run *RTI Data Distribution Service* 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@shmem://
```

(The number 4 is only an example. This is the maximum participant ID.)

9.2 RTI Data Distribution Service Header Files — All Architectures

You must include the appropriate *RTI Data Distribution Service* header files, which are listed in Table 9.1. The header files that need to be included depend on the API being used.

Table 9.1 **Header Files to Include for RTI Data Distribution Service (All Architectures)**

RTI Data Distribution Service API	Header Files
C	#include "ndds/ndds_c.h"
C++	#include "ndds/ndds_cpp.h"
C++/CLI, C#, Java	none

For the compiler to find the included files, the path to the appropriate include directories must be provided. Table 9.2 lists the appropriate include path for use with the compiler. The exact path depends on where you installed *RTI Data Distribution Service*. For example, it may be **C:\Program Files\RTI\ndds.4.5x\include** or **/opt/rti/ndds.4.5x/include** (where *x* stands for the version letter of the current release).

Table 9.2 **Include Paths for Compilation (All Architectures)**

RTI Data Distribution Service API	Include Path Directories
C and C++	<your *RTI Data Distribution Service* installation directory>/include <your *RTI Data Distribution Service* installation directory>/include/ndds
C++/CLI, C#, Java	none

$(NDDSHOME) should be set to the installation directory of *RTI Data Distribution Service* 4.5x, where *x* stands for the version letter of the current release.

The header files that define the data types you want to use within the application also need to be included. For example, Table 9.3 lists the files to be include for type "Foo" (these are the filenames generated by *rtiddsgen*, described in Chapter 3).

Table 9.3 **Header Files to Include for Data Types (All Architectures)**

RTI Data Distribution Service API	User Data Type Header Files
C and C++	#include "Foo.h" #include "FooSupport.h"
C++/CLI, C#, Java	none

9.3 UNIX-based Platforms

Before building an *RTI Data Distribution Service* application for a UNIX-based platform (including Solaris, Red Hat and Yellow Dog Linux, QNX, and LynxOS systems), make sure that:

- ❏ A supported version of your architecture is installed. See the Platform Notes for supported architectures.
- ❏ *RTI Data Distribution Service* 4.4x is installed (where *x* stands for the version letter of the current release). For installation instructions, refer to Section 2.1.1 in the Getting Started Guide.
- ❏ A "make" tool is installed. RTI recommends **gmake** over **make**. If you do not have **gmake**, you can download it from www.sunfreeware.com.

- ❑ The **NDDSHOME** environment variable is set to the root directory of the *RTI Data Distribution Service* installation (such as **/opt/rti/ndds4.5*x***, where *x* stands for the version letter of the current release). To confirm, type this at a command prompt:

  ```
  echo %NDDSHOME%
  env | grep NDDSHOME
  ```

  If it is not set or is set incorrectly, type:

  ```
  setenv NDDSHOME <correct directory>
  ```

To compile an *RTI Data Distribution Service* application of any complexity, either modify the auto-generated makefile created by running **rtiddsgen** or write your own makefile.

9.3.1 Required Libraries

All required system and *RTI Data Distribution Service* libraries are listed in the 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 *RTI Data Distribution Service* applications on the same node. However, shared libraries require more set-up and maintenance during upgrades and installations.

To see if dynamic libraries are supported for your target architecture, see the Platform Notes[1].

9.3.2 Compiler Flags

See the Platform Notes for information on compiler flags.

1. In the *Platform Notes*, see the "Building Instructions..." table for your target architecture.

9.4 Windows Platforms

Before building an application for a Microsoft Windows® platform, make sure that:

- ❑ Supported versions of Windows and Visual C++ or Visual Studio .NET are installed. See Section 9 in the Platform Notes.
- ❑ *RTI Data Distribution Service* 4.5*x* is installed (where *x* stands for the version letter of the current release). For installation instructions, refer to the Section 2.1.2 in the Getting Started Guide.
- ❑ The **NDDSHOME** environment variable is set to the root directory of the *RTI Data Distribution Service* installation (such as **C:\Program Files\RTI\ndds4.5*x***, where ***x*** stands for the version letter of the current release). To confirm, type this at a command prompt:

  ```
  echo %NDDSHOME%
  ```

To compile an *RTI Data Distribution Service* application of any complexity, use a project file in Microsoft Visual Studio. The project settings are described below. Section 9 in the Getting Started Guide contains additional information.

9.4.1 Using Microsoft Visual C++ 6.0

1. From the menu bar, select **Project, Settings...**
2. Select the multi-threaded DLL project setting by following these steps:
   a. Select the **C/C++** tab.
   b. From the **Category** pull-down menu, select **Code Generation**.
   c. From the **Use run-time library** category, select one of the options from Table 9.4.

Table 9.4 **Runtime Library Settings for Visual Studio .NET & Visual Studio .NET 2003**

If You are using this Library Format...	Set the 'Use run-time library' field to...
Release version of static libraries	Multi-threaded
Debug version of static libraries	Debug Multi-threaded
Release version of dynamic libraries	Multi-threaded DLL
Debug version of dynamic libraries	Debug Multi-threaded DLL

3. Link in the *RTI Data Distribution Service* and system libraries:

   a. See Section 9 in the Platform Notes for a list of required libraries. You have a choice of whether to link with *RTI Data Distribution Service*'s static or dynamic libraries. Decide whether or not you want debugging symbols included.

   b. Select the **Link** tab on the **Project Settings** Window.

   c. From the **Category:** pull-down menu, select the **Input** option.

   d. Add the appropriate *RTI Data Distribution Service* and system libraries to the *beginning* of the **Object/Library modules** list. Be sure to use a *space* as a delimiter between libraries, *not* a comma. (Note that some of the system libraries may already be in the list.)

   e. Specify the path to the libraries in the **Additional library path** field by adding the path for your specific architecture. For example:

      ```
      c:\rti\ndds.4.5x\lib\i86Win32VC60
      ```

      Your path may differ, depending on where you installed *RTI Data Distribution Service*. Replace the *x* in 4.5*x* with the version letter for the current release.

4. Specify the path to *RTI Data Distribution Service*'s header files:

   a. Select the **C/C++** tab.

   b. From the **Category** pull-down menu, select the **Preprocessor** option.

   c. In the **Additional include directories** field, add paths to the "**include**" and "**include\ndds**" directories. For example:

      ```
      c:\rti\ndds.4.5x\include\
      c:\rti\ndds.4.5x\include\ndds
      ```

      Your paths may differ, depending on where you installed *RTI Data Distribution Service*. Replace the *x* in 4.5*x* with the version letter for the current release.

5. Specify the compiler flags:

   a. Select the **C/C++** tab.

   b. From the **Category** pull-down menu, select the **Preprocessor** option.

   c. In the **Preprocessor definitions** field, add the compiler flags listed in Table 9.2, "Building Instructions for Windows Host Architectures," on page 3-68. You will see the compiler flag appear in the **Project Options** field.

9.4.2 Using Visual Studio .NET, Visual Studio .NET 2003, or Visual Studio 2005

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 9.5.

Table 9.5 **Runtime Library Settings for Visual Studio .NET, Visual Studio .NET 2003, Visual Studio 2005**

If You are using this Library Format...	Set the Runtime Library field to...
Release version of static libraries	Multi-threaded (/MT)
Debug version of static libraries	Multi-threaded Debug (/MTd)
Release version of dynamic libraries	Multi-threaded DLL (/MD)
Debug version of dynamic libraries	Multi-threaded Debug DLL (/MDd)

2. Link against the *RTI Data Distribution Service* libraries:
   a. Select the **Linker** folder on the Project, Properties dialog box.
   b. Select the **Input** properties.
   c. See Section 9 in the Platform Notes for a list of required libraries. You have a choice of whether to link with *RTI Data Distribution Service*'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 *one* of the following to the **Additional library path** field:

```
$(NDDSHOME)\lib\i86Win32VC70     (for Visual Studio .NET)
$(NDDSHOME)\lib\i86Win32VS2003   (for Visual Studio .NET 2003)
$(NDDSHOME)\lib\i86Win32VS2005   (for Visual Studio .NET 2005)
```

3. Specify the path to *RTI Data Distribution Service*'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:

```
c:\rti\ndds.4.5x\include\
c:\rti\ndds.4.5x\include\ndds
```

Your paths may differ, depending on where you installed *RTI Data Distribution Service*.

9.5 Java Platforms

Before building an application for a Windows or UNIX Java platform, make sure that:

- ❑ *RTI Data Distribution Service* 4.5x is installed (where x stands for the version letter of the current release). For installation instructions, refer to Chapter 2 in the Getting Started Guide.
- ❑ A supported version of the Java 2 software development kit (J2SDK) is installed. See the Platform Notes.

9.5.1 Java Libraries

RTI Data Distribution Service requires that certain Java archive (JAR) files be on your classpath when running *RTI Data Distribution Service* applications. See the Platform Notes for more details.

9.5.2 Native Libraries

RTI Data Distribution Service for Java is implemented using Java Native Interface (JNI), so it is necessary to provide your *RTI Data Distribution Service* distributed applications access to certain native shared libraries. See the Platform Notes for more details.

Part 3: Advanced Concepts

This part of the manual will guide you through some of the more advanced concepts:

- Chapter 10: Reliable Communications
- Chapter 11: Mechanisms for Achieving Information Durability and Persistence
- Chapter 12: Discovery
- Chapter 13: Transport Plugins
- Chapter 14: Built-In Topics
- Chapter 15: Configuring QoS with XML
- Chapter 16: Multi-channel DataWriters
- Chapter 17: RTI Data Distribution Service Threading Model
- Chapter 18: Troubleshooting

Chapter 10 Reliable Communications

RTI Data Distribution Service uses *best-effort* delivery by default. The other type of delivery *RTI Data Distribution Service* supports is called *reliable*. This chapter provides instructions on how to set up and use reliable communication.

This chapter includes the following sections:

❏ Sending Data Reliably (Section 10.1)

❏ Overview of the Reliable Protocol (Section 10.2)

❏ Using QosPolicies to Tune the Reliable Protocol (Section 10.3)

10.1 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 samples of *signal* data can often be dropped because their value disappears when the next sample is sent. However, each 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 samples."
- ❑ **Reliable delivery model** "Make sure all samples get there, in order."

10.1.1 Best-effort Delivery Model

By default, *RTI Data Distribution Service* uses the best-effort delivery model: there is no effort spent ensuring in-order delivery or resending lost samples. Best-effort *DataReaders* ignore lost samples in favor of the latest sample. Your application is only notified if it does not receive a new sample within a certain time period (set in the DEADLINE QosPolicy (Section 6.5.4)).

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 samples continuously. In this situation, a *DataReader* for this *Topic* is only interested in having the latest pressure reading available—older samples are obsolete.

10.1.2 Reliable Delivery Model

Reliable delivery means the samples are guaranteed to arrive, in the order published.

The *DataWriter* maintains a *send queue* with space to hold the last *X* number of samples sent. Similarly, a *DataReader* maintains a *receive queue* with space for consecutive *X* expected samples.

The *send* and *receive queues* are used to temporarily cache samples until *RTI Data Distribution Service* is sure the samples have been delivered and are not needed anymore. *RTI Data Distribution Service* removes samples from a publication's *send queue* after the sample has been acknowledged by all reliable subscriptions. When positive acknowledgements are disabled (see DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2) and DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1)), samples are removed from the send queue after the corresponding keep-duration has elapsed (see Table 6.29, "DDS_RtpsReliableWriterProtocol_t," on page 6-95).

If an out-of-order sample arrives, *RTI Data Distribution Service* speculatively caches it in the *DataReader's receive queue* (provided there is space in the queue). Only consecutive samples are passed on to the *DataReader*.

DataWriters can be set up to wait for available queue space when sending samples. This will cause the sending thread to block until there is space in the *send queue*. (Or, you can decide to sacrifice sending 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 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 samples through its *Listener* and/or *Conditions*.

RTI Data Distribution Service 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 Waiting for Acknowledgments (Section 6.3.10)).

RTI Data Distribution Service 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 *RTI Data Distribution Service* 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 sample has been added to the *DataWriter*'s queue. To see if all *DataReaders* have received the data, look at the RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7) to see if any samples are unacknowledged.

Suppose *DataWriter* A reliably publishes a *Topic* to which *DataReader*s 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 samples that cannot be put into the queue and request A to resend missing samples. The *Listener* is notified with the **on_sample_rejected()** callback (see SAMPLE_REJECTED Status (Section 7.3.6.8)). If A has a queue large enough, or A is no longer writing new samples, A won't notice unless it checks the RELIABLE_WRITER_CACHE_CHANGED Status (DDS Extension) (Section 6.3.5.7).
- ❏ If HISTORY_KEEP_LAST is specified for C, C will drop old samples and accept new ones. The *Listener* is notified with the **on_sample_lost()** callback (see SAMPLE_LOST Status (Section 7.3.6.7)). To A, it is as if all samples have been received by C (that is, they have all been acknowledged).

10. Reliable

10.2 Overview of the Reliable Protocol

An important advantage of *RTI Data Distribution Service* 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. *RTI Data Distribution Service* 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. *RTI Data Distribution Service* 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.[1]

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, liveliness, 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*
- ❑ 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 *RTI Data Distribution Service* 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 10.1 through Figure 10.7, these messages are represented using the notation **DATA(<value>, <sequenceNum>)**. For example,

1. For a link to the RTPS specification, see the RTI website, www.rti.com.

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 10.1 through Figure 10.7 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 *RTI Data Distribution Service* can bundle multiple of the above messages within a single network packet. This 'submessage bundling' provides for higher performance communications.

Figure 10.1 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 10.1 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 10.1 **Basic RTPS Reliable Protocol**

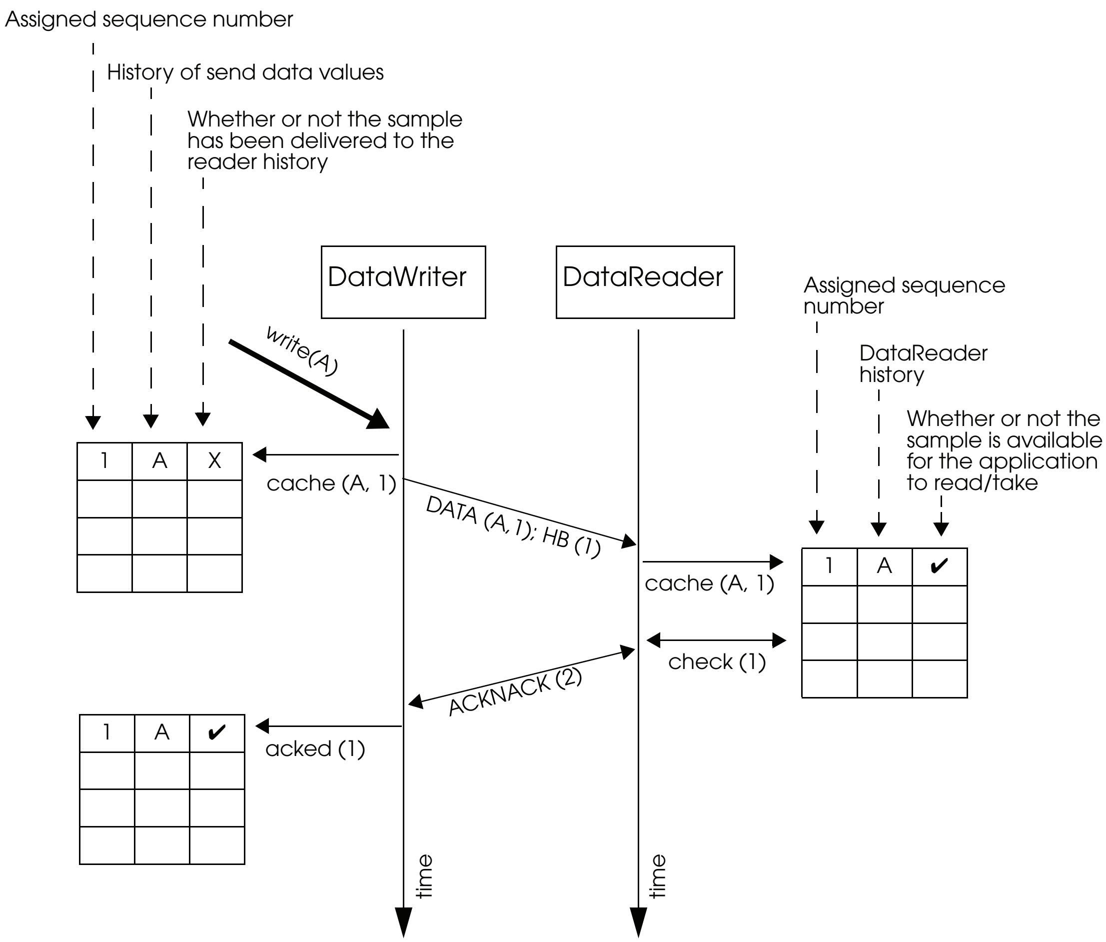
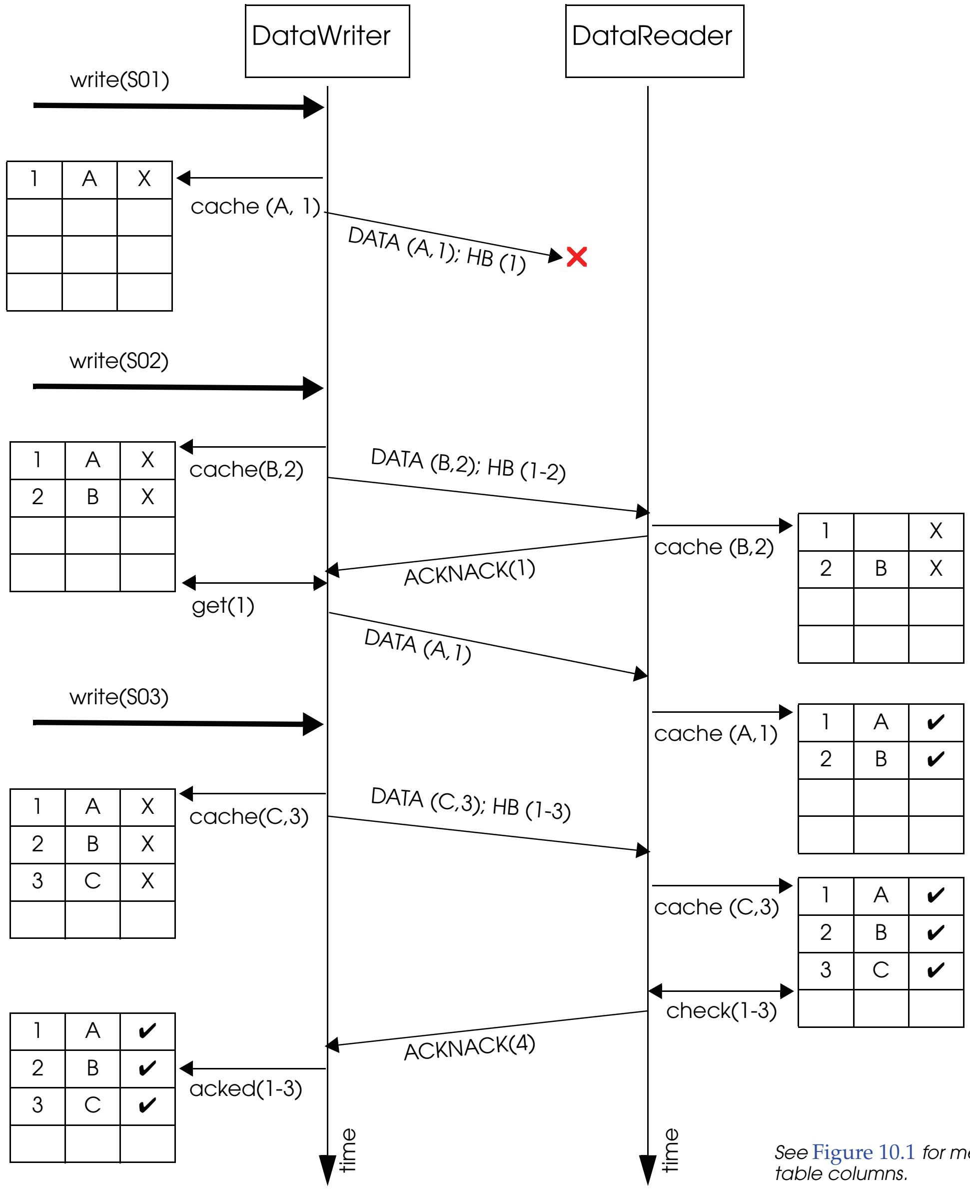


Figure 10.2 **RTPS Reliable Protocol in the Presence of Message Loss**



See Figure 10.1 for meaning of table columns.

Figure 10.2 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:

1. 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 sample 1 (A) first).
2. 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.

10.3 Using QosPolicies to Tune the Reliable Protocol

Reliability is controlled by the QosPolicies in Table 10.1. To enable reliable delivery, read the following sections to learn how to change the QoS for the *DataWriter* and *DataReader*:

- ❑ Enabling Reliability (Section 10.3.1)
- ❑ Tuning Queue Sizes and Other Resource Limits (Section 10.3.2)
- ❑ Controlling Heartbeats and Retries with the DataWriterProtocol QosPolicy (Section 10.3.4)
- ❑ Avoiding Message Storms with DataReaderProtocol QosPolicy (Section 10.3.5)
- ❑ Resending Samples to Late-Joiners with the Durability QosPolicy (Section 10.3.6)

Then see this section to explore example use cases:

- ❑ Use Cases (Section 10.3.7)

Table 10.1 **QosPolicies for Reliable Communications**

QosPolicy	Description	Related Entities[a]	Reference
Reliability	To establish reliable communication, this QoS must be set to **DDS_RELIABLE_RELIABILITY_QOS** for the *DataWriter* and its *DataReaders*.	DW, DR	Section 10.3.1, Section 6.5.17
ResourceLimits	This QoS determines the amount of resources each side can use to manage instances and samples of instances. Therefore it controls the size of the *DataWriter's* send queue and the *DataReader's* receive queue. The send queue stores samples until they have been ACKed by all *DataReaders*. The *DataReader's* receive queue stores samples for the user's application to access.	DW, DR	Section 10.3.2, Section 6.5.18
History	This QoS affects how a *DataWriter*/ *DataReader* behaves when its send/receive queue fills up.	DW, DR	Section 10.3.3, Section 6.5.8

10. Reliable

Table 10.1 **QosPolicies for Reliable Communications**

QosPolicy	Description	Related Entities[a]	Reference
DataWriterProtocol	This QoS configures *DataWriter*-specific protocol. The QoS can disable positive ACKs for its *DataReaders*.	DW	Section 10.3.4, Section 6.5.2
DataReaderProtocol	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*.	DR	Section 10.3.5, Section 7.6.1
DataReader-ResourceLimits	This QoS determines additional amounts of resources that the *DataReader* can use to manage samples (namely, the size of the *DataReader's* receive queue, which stores samples until they are ordered for reliability and can be moved to the *DataReader's* receive queue for access by the user's application).	DR	Section 10.3.2, Section 7.6.2
Durability	This QoS affects whether late-joining *DataReaders* will receive all previously-sent data or not.	DW, DR	Section 10.3.6, Section 6.5.6

a. DW = DataWriter, DR = DataReader

10.3.1 Enabling Reliability

You must modify the RELIABILITY QosPolicy (Section 6.5.17) 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;
```

10.3.1.1 Blocking until the Send Queue Has Space Available

The **max_blocking_time** property in the RELIABILITY QosPolicy (Section 6.5.17) indicates how long a *DataWriter* can be blocked during a **write()**.

If **max_blocking_time** is non-zero and the send queue is full, the write is blocked (the sample is not sent). If **max_blocking_time** has passed and the sample is still not sent, **write()** returns DDS_RETCODE_TIMEOUT and the sample is not sent.

If the number of unacknowledged samples in the send queue drops below **max_samples** (set in the RESOURCE_LIMITS QosPolicy (Section 6.5.18)) before **max_blocking_time**, the sample is sent and **write()** returns **DDS_RETCODE_OK**.

If **max_blocking_time** is zero and the send queue is full, **write()** returns **DDS_RETCODE_TIMEOUT** and the sample is not sent.

10.3.2 Tuning Queue Sizes and Other Resource Limits

Set the HISTORY QosPolicy (Section 6.5.8) appropriately to accommodate however many samples should be saved in the *DataWriter*'s send queue or the *DataReader*'s receive queue. *The defaults may suit your needs;* if so, you do not have to modify this QosPolicy.

For more information, see the following sections:

- Understanding the Send Queue and Setting its Size (Section 10.3.2.1)
- Understanding the Receive Queue and Setting Its Size (Section 10.3.2.2)

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 ResourceLimitsQosPolicy's **max_samples** (see RESOURCE_LIMITS QosPolicy (Section 6.5.18)), and **max_samples_per_remote_writer** (see DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2)) 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*

```
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*

```
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;
```

10.3.2.1 Understanding the Send Queue and Setting its Size

A *DataWriter*'s send queue is used to store each sample it writes. A 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 sample (through an ACKNACK). If that 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 RESOURCE_LIMITS QosPolicy (Section 6.5.18), specifically the **max_samples** field. The appropriate value depends on application parameters such as how fast the publication calls **write()**.

Strict reliability: If a *DataWriter* does not receive ACKNACKs from one or more reliable *DataReader*s, it is possible for the send queue to fill up. If you want to achieve strict reliability, the **kind** field in the HISTORY QosPolicy (Section 6.5.8) 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 queue before writing any more samples. *RTI Data Distribution Service* provides two mechanisms to do this:

- ❑ Allow the **write()** operation to block until there is space in the queue again to store the sample. The maximum time this call blocks is determined by the *max_blocking_time* field in the RELIABILITY QosPolicy (Section 6.5.17) (also discussed in Section 10.3.1.1).
- ❑ Use the *DataWriter*'s *Listener* to be notified when the queue fills up or empties again.

When the HISTORY QosPolicy (Section 6.5.8) on the *DataWriter* is set to KEEP_LAST, strict reliability is not guaranteed. When there are *depth* number of samples in the queue (set in the HISTORY QosPolicy (Section 6.5.8), see Section 10.3.3) the oldest sample will be dropped from the queue when a new sample is written. Note that in such a reliable mode, the *DataWriter* will never block, but strict reliability is no longer guaranteed. If there is a request for the purged sample from any *DataReaders*, the *DataWriter* will send a heartbeat that no longer contains the sequence number of the dropped sample (it will not be able to send the sample).

The send queue size is set in the **max_samples** field of the RESOURCE_LIMITS QosPolicy (Section 6.5.18). The appropriate size for the send queue depends on application parameters (such as the send 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 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 Section 10.3.2.2.

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 send queue, ***N***, is as shown in Figure 10.3.

Figure 10.3 **Calculating Minimum Send Queue Size for a Desired Level of Reliability**

$$N = 2RT\frac{\log(1-Q)}{\log(p)}$$

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 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 sample is eventually successfully delivered. Of course, network-transport dropouts must also be taken into account and may influence or dominate this calculation.

Table 10.2 gives the required size of the send queue for several common scenarios.

Table 10.2 **Required Size of the Send Queue for Different Network Parameters**

Q[a]	p[b]	T[c]	R[d]	N[e]
99%	1%	0.001[f] sec	100 Hz	1
99%	1%	0.001 sec	2000 Hz	2
99%	5%	0.001 sec	100 Hz	1
99%	5%	0.001 sec	2000 Hz	4
99.99%	1%	0.001 sec	100 Hz	1
99.99%	1%	0.001 sec	2000 Hz	6

10. Reliable

Table 10.2 **Required Size of the Send Queue for Different Network Parameters**

Q[a]	p[b]	T[c]	R[d]	N[e]
99.99%	5%	0.001 sec	100 Hz	1
99.99%	5%	0.001 sec	2000 Hz	8

a. "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 samples that will be successfully delivered.

b. "p" is the probability that any single packet gets lost in the network.

c. "T" is the round-trip transport delay in the network

d. "R" is the rate at which the publisher is sending updates.

e. "N" is the minimum required size of the send queue to accomplish the desired level of reliability "Q".

f. The typical round-trip delay for a dedicated 100 Mbit/second ethernet is about 0.001 seconds.

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 10.4.

Figure 10.4 **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 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 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 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.

10.3.2.2 Understanding the Receive Queue and Setting Its Size

Samples are stored in the *DataReader's* receive queue, which is accessible to the user's application.

A sample is removed from the receive queue after it has been accessed by **take()**, as described in Accessing Data Samples with Read or Take (Section 7.4.3). Note that **read()** does not remove samples from the queue.

A *DataReader's* receive queue size is limited by its RESOURCE_LIMITS QosPolicy (Section 6.5.18), specifically the **max_samples** field. The storage of out-of-order samples for each *DataWriter* is also allocated from the *DataReader's* receive queue; this sample resource is shared among all reliable *DataWriters*. That is, **max_samples** includes both ordered and out-of-order samples.

A *DataReader* can maintain reliable communications with multiple *DataWriters* (e.g., in the case of the OWNERSHIP_STRENGTH QosPolicy (Section 6.5.14) setting of SHARED). The maximum number of out-of-order samples from any one *DataWriter* that can occupy in the receive queue is set in the **max_samples_per_remote_writer** field of the DATA_READER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 7.6.2); 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 samples that arrive out of order while waiting for missing samples to be resent. (Up to 256 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 samples and request them again later after the missing 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 sample. However, the receive queue size should generally be larger than the send queue size. Send queue size is discussed in Section 10.3.2.1.

Figure 10.5 and Figure 10.6 compare two hypothetical *DataReaders*, both interacting with the same *DataWriter*. The queue on the left represents an ordering cache, allocated from receive queue—samples are held here if they arrive out of order. The *DataReader* in Figure 10.5 on page 10-16 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 samples are *taken* from the *DataReader* in the *Listener* callback. (See Accessing Data Samples with Read or Take (Section 7.4.3) for information on **take()** and related operations.)

In Figure 10.6 on page 10-17, **max_samples** is too small to cache out-of-order samples for the same operational parameters. In both cases, the *DataReaders* eventually receive all the samples in order. However, the *DataReader* with the larger **max_samples** will get the samples earlier and with fewer transactions. In particular, sample "4" is never resent for the *DataReader* with the larger queue size.

Figure 10.5 **Effect of the Receive-Queue Size on Performance: Large Queue Size**

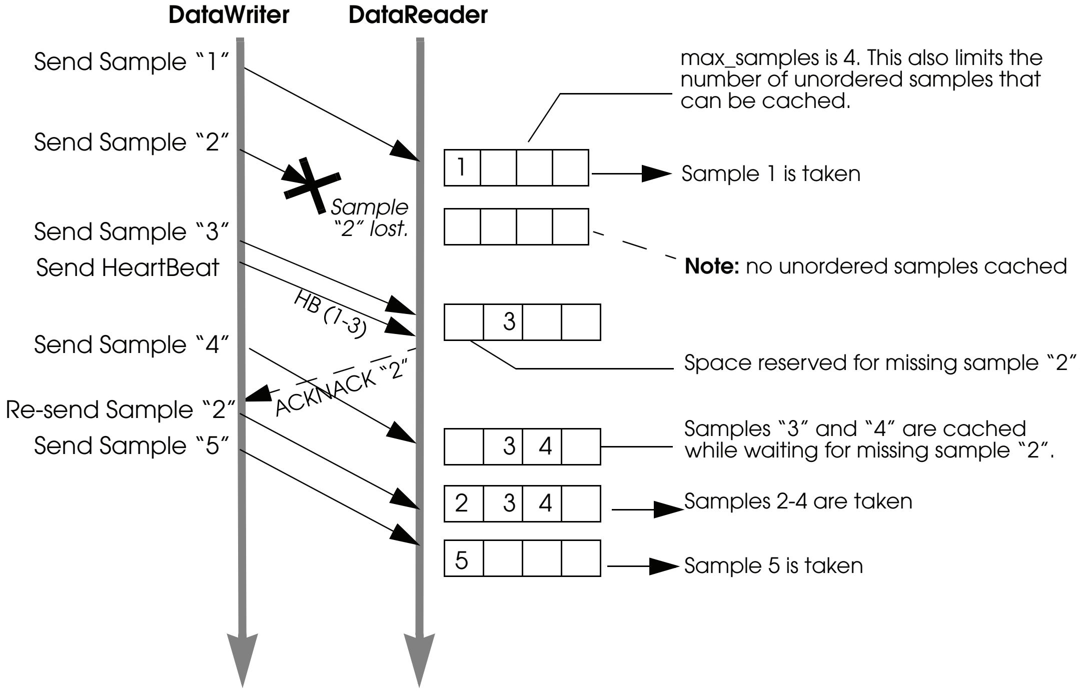
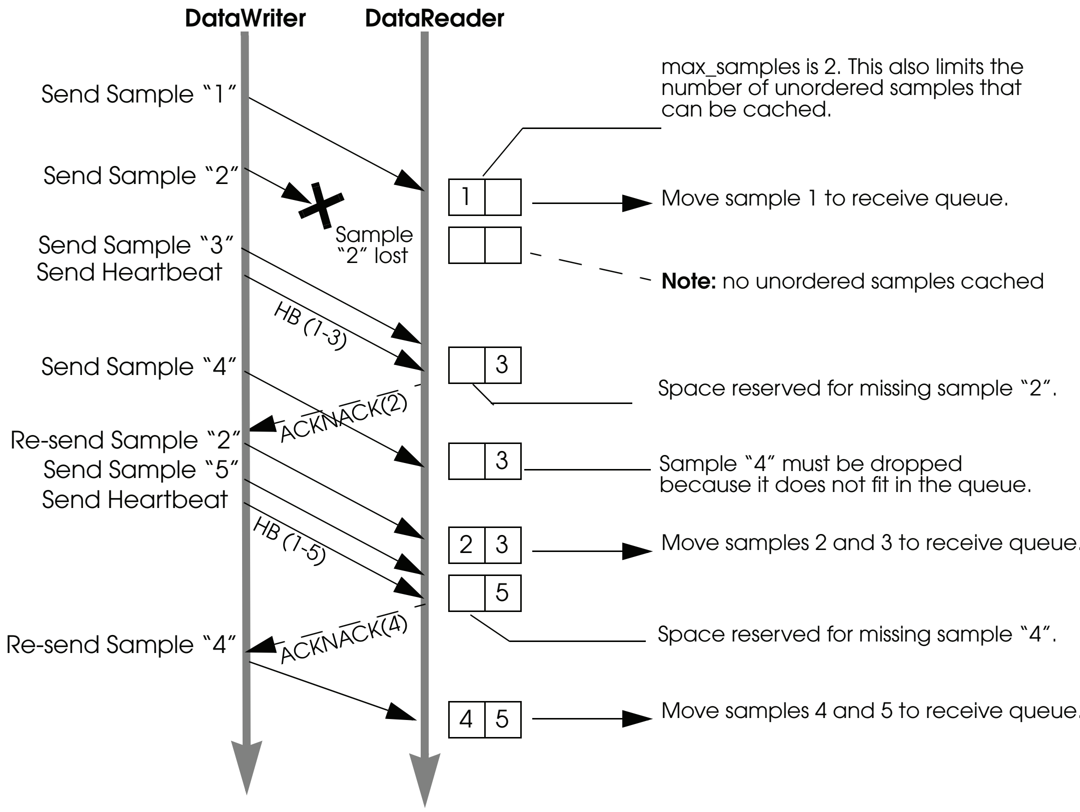


Figure 10.6 **Effect of Receive Queue Size on Performance: Small Queue Size**



10.3.3 Controlling Queue Depth with the History QosPolicy

If you want to achieve strict reliability, set the *kind* field in the HISTORY QosPolicy (Section 6.5.8) 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 samples may not be delivered if the resource limit is reached.

The **depth** field in the HISTORY QosPolicy (Section 6.5.8) controls how many samples *RTI Data Distribution Service* 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 RESOURCE_LIMITS QosPolicy (Section 6.5.18).

Example:

❏ *DataWriter*

```
writer_qos.history.depth =
               <number of samples to keep in send queue>;
```

❏ *DataReader*

```
reader_qos.history.depth =
               <number of samples to keep in receive queue>;
```

10.3.4 Controlling Heartbeats and Retries with the DataWriterProtocol QosPolicy

In the *RTI Data Distribution Service* reliability model, the *DataWriter* sends 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 samples (by sending an ACKNACK) and the *DataWriter* will respond by resending the missing 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 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

The protocol described in Overview of the Reliable Protocol (Section 10.2) 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 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2) and how they affect the behavior of the RTPS protocol.

10.3.4.1 How Often Heartbeats are Resent (heartbeat_period)

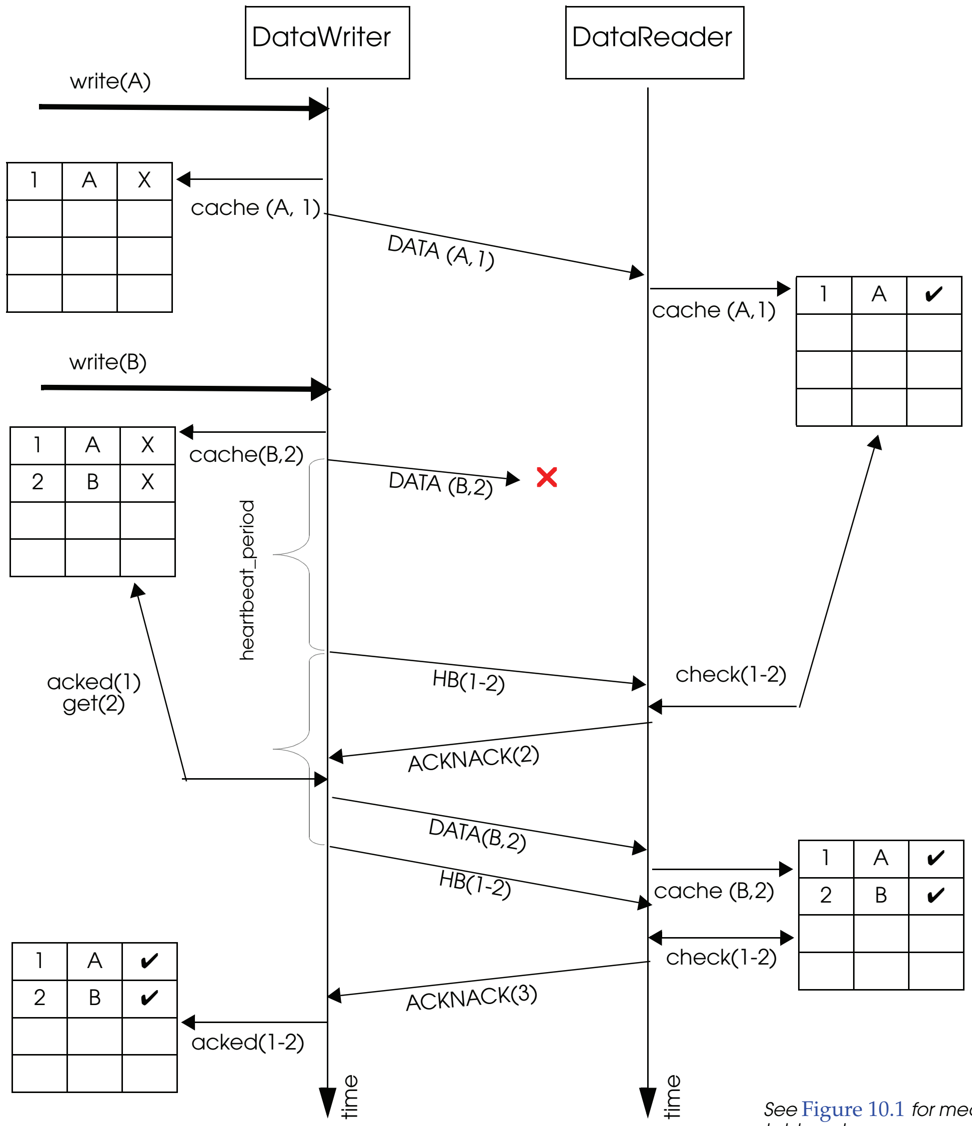
If a *DataReader* does not acknowledge a sample that has been sent, the *DataWriter* resends the heartbeat. These heartbeats are resent at the rate set in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2), specifically its *heartbeat_period* field.

For example, a **heartbeat_period** of 3 seconds means that if a *DataReader* does not receive the latest 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 case shown for message DATA(B, 2) in Figure 10.7. 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, *RTI Data Distribution Service* can be configured so that HBs are sent periodically as long as there are 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 **heartbeat_period** field in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

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.

Figure 10.7 **Use of heartbeat_period**



See Figure 10.1 for meaning of table columns.

10.3.4.2 How Often Piggyback Heartbeats are Sent (heartbeats_per_max_samples)

A *DataWriter* will automatically send heartbeats with new samples to request regular ACKNACKs from the *DataReader*. These are called "piggyback" heartbeats.

If batching is disabled[1]: one piggyback heartbeat will be sent every [**max_samples**[2]/ **heartbeats_per_max_samples**] number of samples.

If batching is enabled: one piggyback heartbeat will be sent every [**max_batches**[3]/ **heartbeats_per_max_samples**] number of samples.

The **heartbeats_per_max_samples** field is part of the **rtps_reliable_writer** structure in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2). If **heartbeats_per_max_samples** is set equal to **max_samples**, this means that a heartbeat will be sent with each sample. A value of 8 means that a heartbeat will be sent with every '*max_samples*/8' samples. Say **max_samples** is set to 1024, then a heartbeat will be sent once every 128 samples. If you set this to zero, samples are sent without any piggyback heartbeat. The **max_samples** field is part of the RESOURCE_LIMITS QosPolicy (Section 6.5.18).

Figure 10.1 on page 10-6 and Figure 10.2 on page 10-7 seem to imply that a 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[4]).
- ❏ 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 data-samples while it waits for ACKNACKs, and the *DataReader*'s receive queue can store out-of-order samples while it waits for missing ones. So it is possible to send HB messages much less frequently than

1. Batching is enabled with the BATCH QosPolicy (DDS Extension) (Section 6.5.1).

2. **max_samples** is set in the RESOURCE_LIMITS QosPolicy (Section 6.5.18)

3. **max_batches** is set in the DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3)

4. 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 (Section 6.5.8) or LIFESPAN (Section 6.5.10).

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 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

A HB is used to get confirmation from *DataReaders* so that the *DataWriter* can remove acknowledged samples from the queue to make space for new samples. Therefore, if the queue size is large, or new samples are added slowly, HBs can be sent less frequently.

In Figure 10.8 on page 10-23, the *DataWriter* sets the **heartbeats_per_max_samples** to certain value so that a piggyback HB will be sent for every three samples. The *DataWriter* first writes 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 samples in its queue. When the *DataWriter* sends sample C, it will send a piggyback HB along with the sample. Once the *DataReader* receives the HB, it will send back an ACKNACK for samples up to sequence number 3, such that the *DataWriter* can remove all three samples from its queue. ,

10.3.4.3 Controlling Packet Size for Resent Samples (max_bytes_per_nack_response)

A repair packet is the maximum amount of data that a *DataWriter* will resend at a time. For example, if the *DataReader* requests 20 samples, each 10K, and the **max_bytes_per_nack_response** is set to 100K, the *DataWriter* will only send the first 10 samples. The *DataReader* will have to ACKNACK again to receive the next 10 samples.

A *DataWriter* may resend multiple missed samples in the same packet. The **max_bytes_per_nack_response** field in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2) limits the size of this 'repair' packet.

10.3.4.4 Controlling How Many Times Heartbeats are Resent (max_heartbeat_retries)

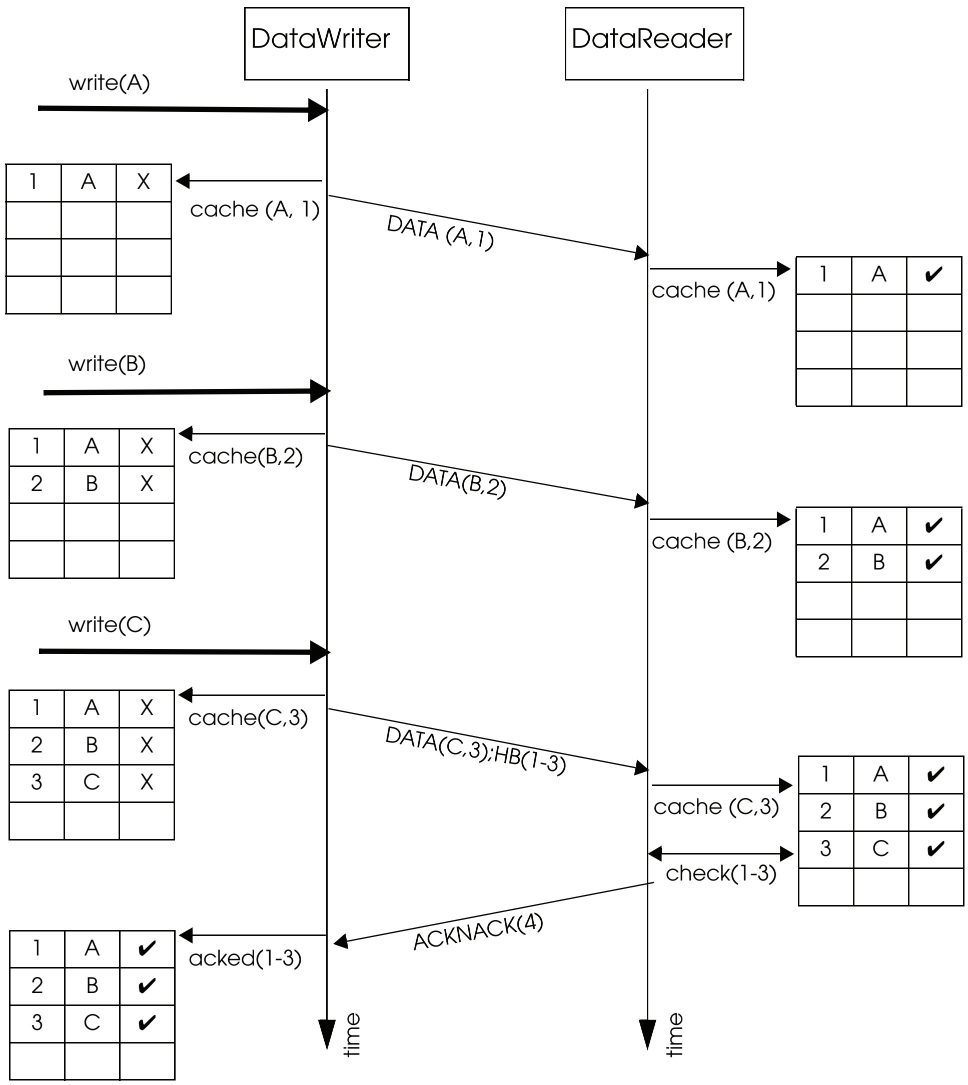
If a *DataReader* does not respond within **max_heartbeat_retries** number of heartbeats, it will be dropped by the *DataWriter* and the reliable *DataWriter*'s *Listener* will be called with a RELIABLE_READER_ACTIVITY_CHANGED Status (DDS Extension) (Section 6.3.5.8).

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 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 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

Figure 10.8 **Use of the heartbeats_per_max_samples**



See Figure 10.1 for meaning of table columns.

10.3.4.5 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 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 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*.

10.3.4.6 Coping with Redundant Requests for Missing Samples (max_nack_response_delay)

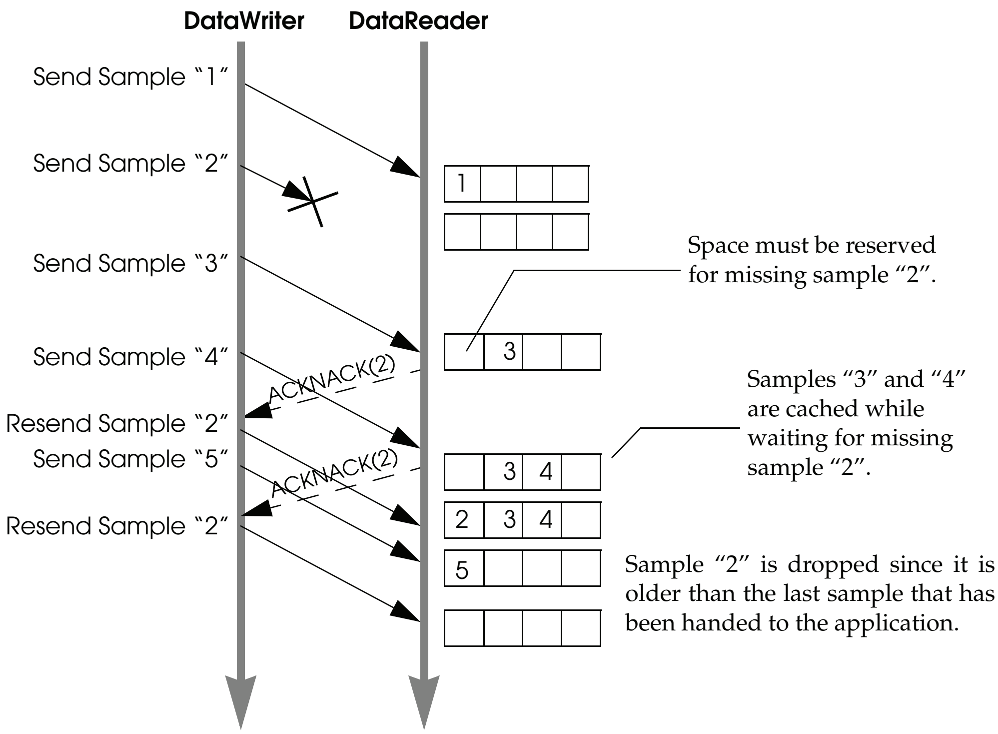
When a *DataWriter* receives a request for missing samples from a *DataReader* and responds by resending the requested samples, it will ignore additional requests for the same samples during the time period **max_nack_response_delay**.

The **rtps_reliable_writer.max_nack_response_delay** field is part of the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

If your send period is smaller than the round-trip delay of a message, this can cause unnecessary sample retransmissions due to redundant ACKNACKs. In this situation, an ACKNACK triggered by an out-of-order sample is not received before the next sample is sent. When a *DataReader* receives the next message, it will send another ACKNACK for the missing sample. As illustrated in Figure 10.9 on page 10-25, duplicate ACKNACK messages cause another resending of missing sample “2” and lead to wasted CPU usage on both the publication and the subscription sides.

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**.

Figure 10.9 **Resending Missing Samples due to Duplicate ACKNACKs**



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 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 sample inherently takes a long time in this system. So you should allow a longer time for recovery of the lost 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 sample is more important than conserving CPU and network bandwidth (new samples that are too far ahead of the missing sample are thrown away). A fast system can cope with a smaller **min_nack_response_delay** value, and the reliable sample stream can normalize more quickly.

10.3.4.7 Disabling Positive Acknowledgements (disable_postive_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 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 sample for a sufficient duration for ACK-disabled *DataReaders* to have a chance to NACK it. Thus, a configurable “keep-duration” (**disable_postive_acks_min_sample_keep_duration**) applies for samples written for ACK-disabled *DataReaders*, where samples are kept in the queue for at least that keep-duration. After the keep-duration has elapsed for a sample, the 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 sample is written to when a NACK for the sample could be received. If set too short, the 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.

10.3.5 Avoiding Message Storms with DataReaderProtocol QosPolicy

DataWriters send 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 DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1).

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:

```
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 samples.

10.3.6 Resending Samples to Late-Joiners with the Durability QosPolicy

The DURABILITY QosPolicy (Section 6.5.6) is also somewhat related to Reliability. *RTI Data Distribution Service* 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 sample will not actually get sent. Whether or not samples are resent when the *DataReader* and *DataWriter* eventually "discover" one another depends on how the DURABILITY and HISTORY QoS are set. The default setting for the Durability QosPolicy is VOLATILE, which means that the *DataWriter* will *not* store samples for redelivery to late-joining *DataReaders*.

RTI Data Distribution Service also supports the TRANSIENT_LOCAL setting for the Durability, which means that the samples will be kept stored for redelivery to late-joining *DataReaders*, as long as the *DataWriter* is around and the RESOURCE_LIMITS QosPolicy (Section 6.5.18) allows. The samples are not stored beyond the lifecycle of the *DataWriter*.

See also: Waiting for Historical Data (Section 7.3.5).

10.3.7 Use Cases

This section contains advanced material that discusses practical applications of the reliability related QoS.

10.3.7.1 Importance of Relative Thread Priorities

For high throughput, the *RTI Data Distribution Service* Event thread's priority must be sufficiently high on the sending application. Unlike an unreliable writer, a reliable writer relies on internal *RTI Data Distribution Service* threads: the Receive thread processes ACKNACKs from the *DataReaders*, and the Event thread schedules the events necessary to maintain reliable data flow.

- ❑ When 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 DDSDataWriter). 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 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 *RTI Data Distribution Service* 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 EVENT QosPolicy (DDS Extension) (Section 8.5.6)).

  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:

- ❑ Aperiodic Use Case: One-at-a-Time (Section 10.3.7.2)
- ❑ Aperiodic, Bursty (Section 10.3.7.3)
- ❑ Periodic (Section 10.3.7.4)

10.3.7.2 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 sample to guarantee reception of the just sent sample on the reader's middleware end, a smaller queue will provide a smaller upper bound on the sample delivery time. Adequate writer QoS for this use case are presented in Figure 10.10.

Line 1 (Figure 10.10): This is the default setting for a writer, shown here strictly for clarity.

Line 2 (Figure 10.10): Setting the History kind to KEEP_ALL guarantees that no sample is ever lost.

Line 3 (Figure 10.10): 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 sample loss. (See DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).) 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 10.10): For this example, we assume a single writer is writing samples one at a time. If we are not using keys (see Section 2.2.2), there is no reason to use a queue with room for more than one sample, because we want to resolve a sample completely before moving on to the next. While this negatively impacts throughput, it minimizes memory usage. In this example, a written sample will remain in the queue until it is acknowledged by all active readers (only 1 for this example).

Line 12 - Line 14 (Figure 10.10): The fastest way for a writer to ensure that a reader is up-to-date is to force an acknowledgement with every sample. We do this by appending a Heartbeat with every 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 10.10): Since the writer takes responsibility for pushing the 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 sample is written for this writer) and only

Figure 10.10 **QoS for an Aperiodic, One-at-a-time 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
5    //use these hard coded value unless you use a key
6    qos->resource_limits.initial_samples = qos->resource_limits.max_samples = 1;
7    qos->resource_limits.max_samples_per_instance =
8        qos->resource_limits.max_samples;
9    qos->resource_limits.initial_instances =
10       qos->resource_limits.max_instances = 1;
11
12   // want to piggyback HB w/ every sample.
13   qos->protocol.rtps_reliable_writer.heartbeats_per_max_samples =
14       qos->resource_limits.max_samples;
15
16   qos->protocol.rtps_reliable_writer.high_watermark = 1;
17   qos->protocol.rtps_reliable_writer.low_watermark = 0;
18   qos->protocol.rtps_reliable_writer.min_nack_response_delay.sec = 0;
19   qos->protocol.rtps_reliable_writer.min_nack_response_delay.nanosec = 0;
20   //consider making non-zero for reliable multicast
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
24   // should be faster than the send rate, but be mindful of OS resolution
25   qos->protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;
26   qos->protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec =
27           alertReaderWithinThisMs * 1000000;
28
29   qos->reliability.max_blocking_time = blockingTime;
30   qos->protocol.rtps_reliable_writer.max_heartbeat_retries = 7;
31
32   // essentially turn off slow HB period
33   qos->protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;
```

come out of this mode when the low water mark is reached (when all 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 10.10): When a reader requests a lost 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 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 DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2) for information on setting **min_nack_response_delay** and **max_nack_response_delay**.) Note that *RTI Data Distribution Service* 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 millisecond). 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 10.10): If a reader drops a sample, the writer recovers by notifying the reader of what it has sent, so that the reader may request resending of the lost 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 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 DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1) for how to change this). If a sample is dropped, 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 sample immediately, and with the code set as above, the writer will feed the missing sample immediately. Neglecting the processing time on the writer or the reader end, and assuming that this retry succeeds, the time to recover the 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 * alertReaderWithinThisMs + 2 * 50 µsec + 25 µsec**. If **alertReaderWithinThisMs** is 100 ms, the recovery time now exceeds 200 ms, and can perhaps degrade user experience.

Line 29-Line 30 (Figure 10.10): What if another command (like another button press) is issued before the recovery? Since we must not drop this new 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 samples are irrecoverable. So estimating the worst case unresponsive time of all potential readers is critical if sample drop is unacceptable.

Line 32-Line 33 (Figure 10.10): 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 10.11 shows how to set the QoS for the reader side, followed by a line-by-line explanation.

Figure 10.11 **QoS for an Aperiodic, One-at-a-time Reliable Reader**

```
qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;

// 1 is ok for normal use.  2 allows fast infinite loop
qos->reader_resource_limits.max_samples_per_remote_writer = 2;
qos->resource_limits.initial_samples = 2;
qos->resource_limits.initial_instances = 1;

qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec = 0;
qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;
```

Line 1-Line 2 (Figure 10.11): 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 10.11): Since we enforce reliability on each 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 *RTI Data Distribution Service* passes the user data up to the application even before acknowledging the sample to the writer (for minimum latency), the first sample is still pending for acknowledgement in the writer's queue when the writer attempts to write the second sample, and will cause the writing thread to block until the reader completes processing the first 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 sample. Let's follow the packets flow under a normal circumstance:

1. The sender application writes 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 sample 1.
2. The sender application writes sample 2 to the receiving application in response to response 1. Because the reader's queue is 2, it can accept sample 2 even though it may not yet have acknowledged sample 1. Otherwise, the reader may drop sample 2, and would have to recover it later.
3. At the same time, the receiver application acknowledges sample 1, and frees up one slot in the queue, so that it can accept sample 3, which it on its way.

The above steps can be repeated ad-infinitum in a continuous traffic.

Line 7 (Figure 10.11): Since we are not using keys, there is just one instance.

Line 9-Line 12 (Figure 10.11): 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.

10.3.7.3 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 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 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.

RTI Data Distribution Service 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 (Section 10.3.7.2), 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 10.12, with a line-by-line explanation below.

Line 1 (Figure 10.12): This is the default setting for a writer, shown here strictly for clarity.

Line 2 (Figure 10.12): Since we do not want any data lost, we want the History kind set to KEEP_ALL.

Line 3 (Figure 10.12): The default *RTI Data Distribution Service* reliable writer will push, but we want the reader to pull instead.

Line 5-Line 11 (Figure 10.12): We assume a single instance, in which case the maximum sample count will be the same as the maximum sample count per writer. In contrast to the one-at-a-time case discussed in Section 10.3.7.2, 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 10.12): By a 'piggyback' 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 sample request, but we call that a 'framing' Heartbeat. Since data is pulled, **heartbeats_per_max_samples** is ignored.

Line 16-Line 17 (Figure 10.12): Similar to the previous aperiodic writer, this writer spends most of its time idle. But as the name suggests, even a single new sample implies more sample to follow in a burst. Putting the writer into a fast mode quickly will allow

Figure 10.12 **QoS for an Aperiodic, Bursty 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_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      alertReaderWithinThisMs * 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;
```


readers to be notified soon. Only when all samples have been delivered, the writer can rest.

Line 19- Line 23 (Figure 10.12): 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 samples at a time without being overwhelmed, and that the time it takes a writer to fetch these 20 samples and send a single packet containing these 20 samples, plus the time it takes a reader to receive and process these sample samples, and send another request back to the writer for the next 20 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 *RTI Data Distribution Service* 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 samples. The writing thread will of course not block because it is merely copying each of the 1000 samples to the *RTI Data Distribution Service* queue on the writer side; on a typical modern machine, the act of writing these 1000 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, *RTI Data Distribution Service* supports multicast by measuring how many multicast readers have requested 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 Section 10.3.7.2, 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 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 (Figure 10.12): 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 10.12): 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 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 10.12): For an aperiodic writer, turning off slow periodic Heartbeats will remove unwanted traffic from the network.

Figure 10.13 shows example code for a corresponding aperiodic, bursty reader.

Figure 10.13 **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
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
13  // the writer probably has more for the reader; ask right away
14  qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
15  qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;
16  qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
17  qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec = 0;
```

Line 1-Line 2 (Figure 10.13): 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 10.13): 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 sample, it will only ask for this much.

Line 7-Line 11 (Figure 10.13): We do not use keys in this example.

Line 13-Line 17 (Figure 10.13): 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.

10.3.7.4 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 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 10.14 shows an example, with line-by-line analysis below.

Figure 10.14 **QoS for a Periodic Reliable Writer**

```
qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
qos->protocol.push_on_write = DDS_BOOLEAN_TRUE;

//use these hard coded value until you use key
qos->resource_limits.initial_instances =
    qos->resource_limits.max_instances = 1;

int unresolvedSamplePerRemoteWriterMax =
    worstCaseApplicationDelayTimeInMs * dataRateInHz / 1000;
qos->resource_limits.max_samples = unresolvedSamplePerRemoteWriterMax;
qos->resource_limits.initial_samples = qos->resource_limits.max_samples/2;
qos->resource_limits.max_samples_per_instance =
    qos->resource_limits.max_samples;

int piggybackEvery = 8;
qos->protocol.rtps_reliable_writer.heartbeats_per_max_samples =
    qos->resource_limits.max_samples / piggybackEvery;

qos->protocol.rtps_reliable_writer.high_watermark = piggybackEvery * 4;
qos->protocol.rtps_reliable_writer.low_watermark = piggybackEvery * 2;
qos->reliability.max_blocking_time = blockingTime;

qos->protocol.rtps_reliable_writer.min_nack_response_delay.sec = 0;
qos->protocol.rtps_reliable_writer.min_nack_response_delay.nanosec = 0;

qos->protocol.rtps_reliable_writer.max_nack_response_delay.sec = 0;
qos->protocol.rtps_reliable_writer.max_nack_response_delay.nanosec = 0;

qos->protocol.rtps_reliable_writer.fast_heartbeat_period.sec = 0;
qos->protocol.rtps_reliable_writer.fast_heartbeat_period.nanosec =
    alertReaderWithinThisMs * 1000000;
qos->protocol.rtps_reliable_writer.max_heartbeat_retries = 7;

// essentially turn off slow HB period
qos->protocol.rtps_reliable_writer.heartbeat_period.sec = 3600 * 24 * 7;
```

Line 1 (Figure 10.14): This is the default setting for a writer, shown here strictly for clarity.

Line 2 (Figure 10.14): Since we do not want any data lost, we set the History kind to KEEP_ALL.

Line 3 (Figure 10.14): 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 10.14): We do not use keys in this example, so there is only one instance.

Line 9-Line 11 (Figure 10.14): 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 wort case jitter in communication.

Line 12 (Figure 10.14): 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, *RTI Data Distribution Service* will keep increasing the queue size as necessary to accommodate new 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) * sizeof(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 10.14): If there is only one instance, maximum samples per instance is the same as maximum samples allowed.

Line 16-Line 18 (Figure 10.14): 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 samples. The **piggybackEvery** can be increased if the writer is writing at a fast rate, with the cost that more 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 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 samples.

Line 20-Line 22 (Figure 10.14): 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** 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 10.14): 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 10.14): 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 10.14): Since this a periodic model, a separate periodic heartbeat to notify the writer's status would seem unwarranted; the piggyback heartbeat sent with samples takes over that role.

Figure 10.15 shows how to set the QoS for a matching reader, followed by a line-by-line explanation.

Figure 10.15 **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
8   //use these hard coded value until you use key
9   qos->resource_limits.max_samples_per_instance =
10      qos->resource_limits.max_samples;
11  qos->resource_limits.initial_instances =
12     qos->resource_limits.max_instances = 1;
13
14  qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.sec = 0;
15  qos->protocol.rtps_reliable_reader.min_heartbeat_response_delay.nanosec = 0;
16  qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.sec = 0;
17  qos->protocol.rtps_reliable_reader.max_heartbeat_response_delay.nanosec = 0;
```

Line 1-Line 2 (Figure 10.15): 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 10.15) Unlike the writer, the reader queue is sized not according to the jitter of the reader, but rather how many samples you want to cache speculatively in case of a gap in sequence of samples that the reader must recover. Remember that a reader will stop giving a sequence of 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 samples received until the one being sought is recovered. *RTI Data Distribution Service* uses speculative caching, which minimizes the disruption caused by a few dropped 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 2 factors that comprise the lost 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 samples between two piggybacked samples are dropped, the reader must cache **piggybackEvery** samples before asking the writer for resend. But if a piggybacked sample is also lost, the reader will not get around to asking the writer until the next piggybacked 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*piggybackEvery - 1**) samples before the reader will ask the writer to resend, assuming no delay (Line 14-Line 17).

- 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 10.15): Since we are not using keys, there is just one instance.

Line 14-Line 17 (Figure 10.15): If we are not using multicast, or the number of readers being fed by the writer, there is no reason to delay.

Chapter 11 Mechanisms for Achieving Information Durability and Persistence

11.1 Introduction

RTI Data Distribution Service 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 DURABILITY QosPolicy (Section 6.5.6) 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, you need *RTI Persistence Service*, a separate application described in Chapter 21: Introduction to RTI Persistence Service.

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 *RTI Data Distribution Service*. Supported databases are listed in the *Release Notes*. *RTI Persistence Service* does not require a database when used in TRANSIENT mode (see Section 11.5.1) or in PERSISTENT mode with file-system storage (see Section 11.5.1 and Section 22.5).

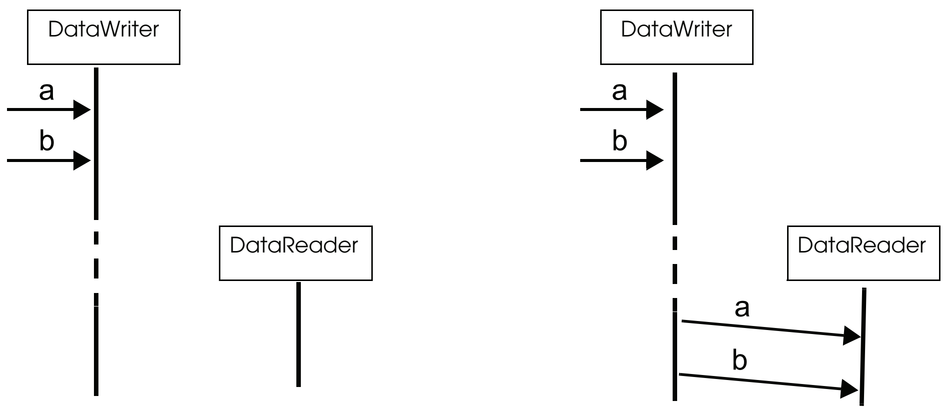
To understand how these features interact we will examine the behavior of the system using the following scenarios:

- ❏ Scenario 1. A DataReader Joins after a DataWriter Restarts (Durable Writer History) (Section 11.1.1)
- ❏ Scenario 2: A DataReader Restarts While DataWriter Stays Up (Durable Reader State) (Section 11.1.2)
- ❏ Scenario 3. A DataReader Joins after the DataWriter Leaves the Domain (Durable Data) (Section 11.1.3)

11.1.1 Scenario 1. A DataReader Joins after a 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 11.1. For more information, see Durable Writer History (Section 11.3)

Figure 11.1 **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.

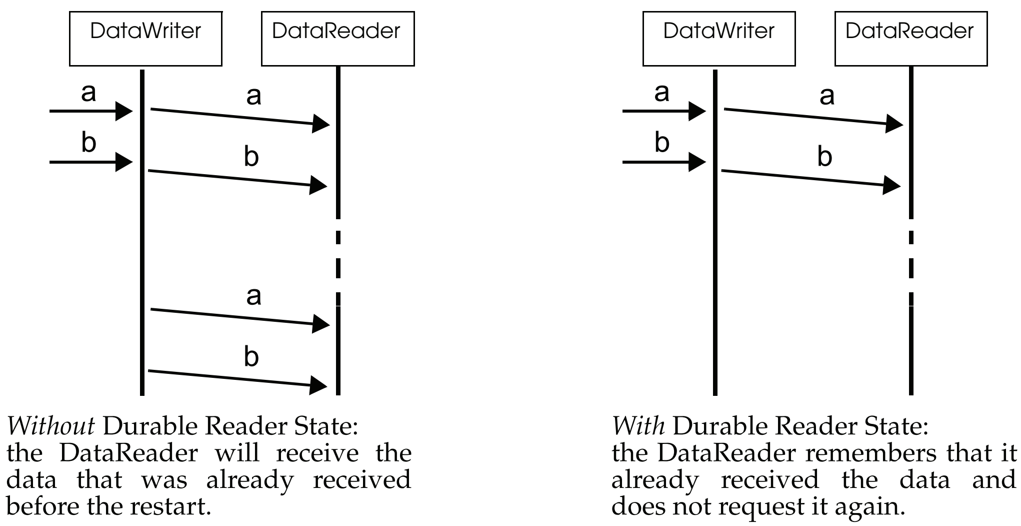
With Durable Writer History: the restarted DataWriter will recover its history and deliver its data to the late-joining DataReader

11.1.2 Scenario 2: A 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 11.2. For more information, see Durable Reader State (Section 11.4).

Figure 11.2 **Durable Reader State**



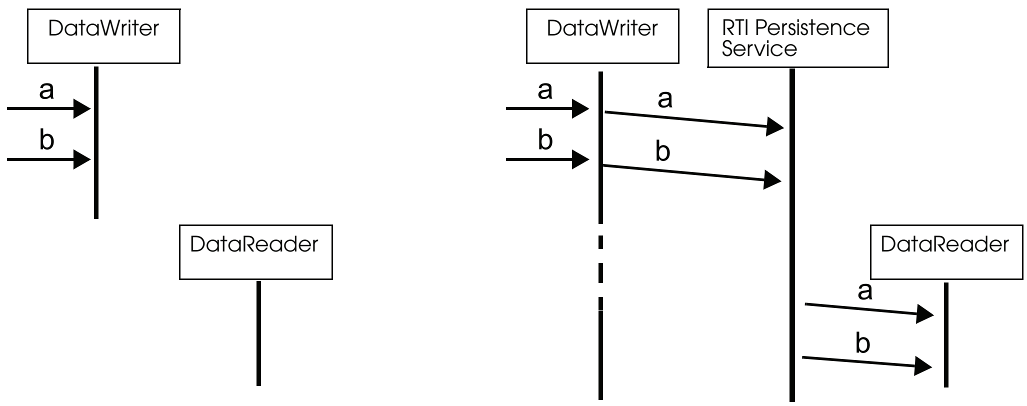
11.1.3 Scenario 3. A DataReader Joins after the DataWriter Leaves the Domain (Durable Data)

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. *RTI 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 11.3. For more information, see Data Durability (Section 11.5)

Figure 11.3 **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:
RTI Persistence Service remembers what data was published and delivers it to the late-joining *DataReader*.

This third scenario is similar to Scenario 1. A DataReader Joins after a DataWriter Restarts (Durable Writer History) (Section 11.1.1) 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 *RTI Persistence Service* acts as an intermediary that stores the data so it can be given to late-joining *DataReaders*.

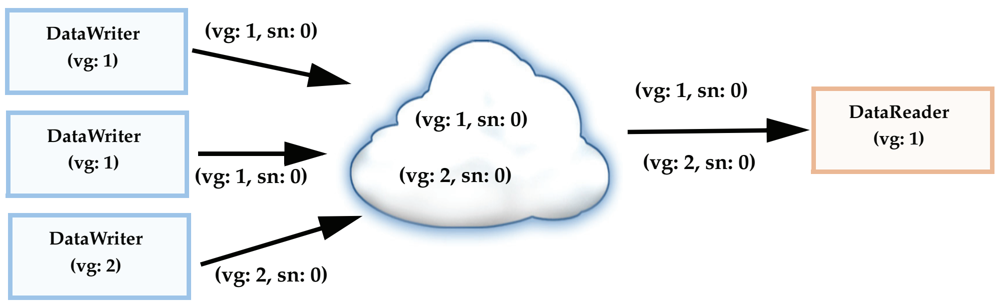
11.2 Durability and Persistence Based on Virtual GUIDs

Every modification to the global dataspace made by a *DataWriter* is identified by a pair (virtual GUID, sequence number).

- ❑ 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 11.4).

Figure 11.4 **Global Dataspace Changes**

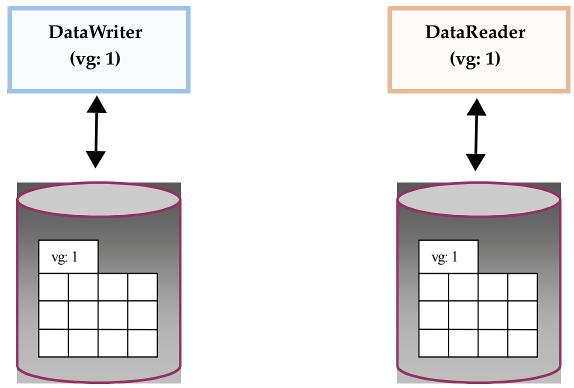


Additionally, *RTI Data Distribution Service* uses the virtual GUID to associate a persisted state (state in permanent storage) to the corresponding DDS entity.

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 11.5).

Figure 11.5 **History/State Persistence Based on the Virtual GUID**



A *DataWriter*'s virtual GUID can be configured using the member **virtual_guid** in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2).

A *DataReader's* virtual GUID can be configured using the member **virtual_guid** in the DATA_READER_PROTOCOL QosPolicy (DDS Extension) (Section 7.6.1).

The DDS_PublicationBuiltinTopicData and DDS_SubscriptionBuiltinTopicData structures include the virtual GUID associated with the discovered publication or subscription (see Built-in DataReaders (Section 14.2)).

11.3 Durable Writer History

The DURABILITY QosPolicy (Section 6.5.6) 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.

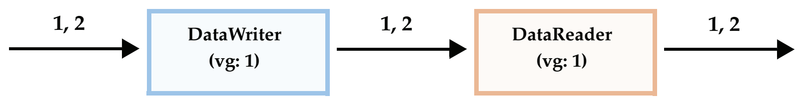
RTI Data Distribution Service provides the capability to make the *DataWriter* history durable, by persisting its content in a relational database. This makes it possible for the history to be restored when the *DataWriter* restarts. See the Release Notes for the list of supported relational databases.

The association between the history stored in the database and the *DataWriter* is done using the virtual GUID.

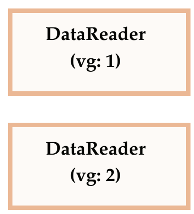
11.3.1 Durable Writer History Use Case

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.

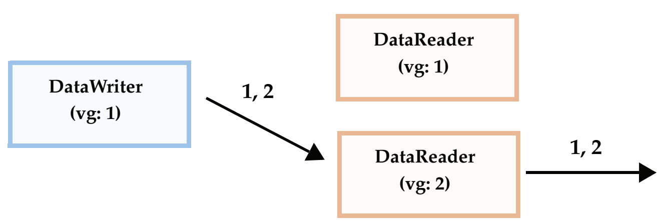


2. The process running the *DataWriter* is stopped and a new late-joining *DataReader* is created.



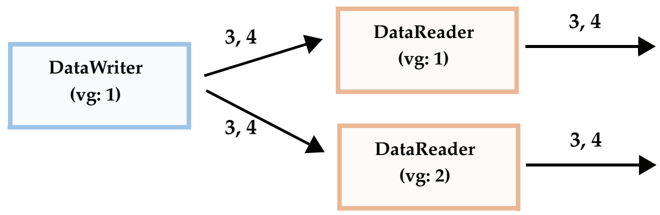
   The new *DataReader* with virtual GUID 2 does not receive samples 1 and 2 because the original *DataWriter* has been destroyed. If the samples must be available to late-joining *DataReaders* after the *DataWriter* deletion, you can use *RTI Persistence Service*, described in Chapter 21: Introduction to RTI Persistence Service.

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*.

11.3.2 How To Configure Durable Writer History

RTI Data Distribution Service 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, *RTI Data Distribution Service* 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 PROPERTY QosPolicy (DDS Extension) (Section 6.5.15) 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 11.1 lists the supported 'durable writer history' properties.

Note: Durable Writer History is not supported for Multi-channel *DataWriters* (see Chapter 16) or when Batching is enabled (see Section 6.5.1); an error is reported if this type of *DataWriter* tries to configure Durable Writer History.

See also: Durable Reader State (Section 11.4).

Table 11.1 **Durable Writer History Properties**

Property	Description
dds.data_writer.history.plugin_name	**Required.** Must be set to **"dds.data_writer.history.odbc_plugin.builtin"** to enable durable writer history in the *DataWriter*.
dds.data_writer.history.odbc_plugin.dsn	**Required.** The ODBC DSN (Data Source Name) associated with the database where the writer history must be persisted.
dds.data_writer.history.odbc_plugin.driver	This property tells *RTI Data Distribution Service* which ODBC driver to load. If the property is not specified, *RTI Data Distribution Service* will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems). **When using Durable Writer History and Oracle TimesTen on a Windows system:** You must link directly with the TimesTen driver, not with the Windows ODBC Driver Manager. To do so, you must set **dds.data_writer.history.odbc_plugin.driver** to **ttdv60.dll** (this is not the default).
dds.data_writer.history.odbc_plugin.username	Configures the username/password used to connect to the database. Default: No password or username
dds.data_writer.history.odbc_plugin.password	
dds.data_writer.history.odbc_plugin.shared	When set to 1, *RTI Data Distribution Service* 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).

Table 11.1 **Durable Writer History Properties**

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>dds.data_writer.history.odbc_plugin.
instance_cache_max_size</td><td rowspan="4">These properties configure the resource limits associated with the ODBC writer history caches.
To minimize the number of accesses to the database, RTI Data Distribution Service 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.
The resource limits, initial_instances, max_instances, initial_samples, max_samples, and max_samples_per_instance defined in RESOURCE_LIMITS QosPolicy (Section 6.5.18) are used to configure the maximum number of samples and instances that can be stored in the relational database.
Defaults:
❑ instance_cache_max_size: max_instances in RESOURCE_LIMITS QosPolicy (Section 6.5.18)
❑ instance_cache_init_size: initial_instances in RESOURCE_LIMITS QosPolicy (Section 6.5.18)
❑ sample_cache_max_size: 32
❑ sample_cache_init_size: 32
Note: If the property in_memory_state (see below in this table) is 1, then instance_cache_max_size is always equal to max_instances in RESOURCE_LIMITS QosPolicy (Section 6.5.18)—it cannot be changed.</td></tr>
<tr><td>dds.data_writer.history.odbc_plugin.
instance_cache_init_size</td></tr>
<tr><td>dds.data_writer.history.odbc_plugin.
sample_cache_max_size</td></tr>
<tr><td>dds.data_writer.history.odbc_plugin.
sample_cache_init_size</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>
</table>

Table 11.1 **Durable Writer History Properties**

Property	Description
dds.data_writer.history.odbc_plugin.in_memory_state	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 **instance_cache_max_size** (see above in this table) is always equal to **max_instances** in RESOURCE_LIMITS QosPolicy (Section 6.5.18)—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 unless the underlying database is an in-memory database (TimesTen). Default: 1

Example C++ Code

```
/* 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);

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 */
}

/* Create Data Writer */
...
```

11.4 Durable Reader State

Durable reader state allows a *DataReader* to locally persists 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. Data that was already received by the *DataReader* before the restart will be suppressed so it is not sent over the network again.

RTI Data Distribution Service provides the capability to persist the state of a *DataReader* in a relational database. This database is accessed using ODBC. See the Release Notes for the list of supported relational databases.

For each *DataReader* that is configured to be durable, *RTI Data Distribution Service* will create one database table with the following name RS<32 uuencoding of the reader virtual GUID>. This table will be used to store the last sample (sequence number) received from each virtual *DataWriter*. For *DataReaders* on keyed topics requesting instance-ordering (see PRESENTATION QosPolicy (Section 6.4.6)), this state will be stored per instance per virtual *DataWriter*, not simply per virtual *DataWriter*.

For the read/take methods that require calling **return_loan()**, a sample 's1' with sequence number 'sn1' sent by a *DataWriter* 'dw1' will be considered as received when the application return the loan on 'sn1' or any other sample sent by the same *DataWriter* (same virtual GUID) with a sequence number equal or greater than 'sn1'. For example:

```
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) {
```

11. Durability and Persistence

```
        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 consid-
    ered 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 'sn1' sent by a *DataWriter* 'dw1' will be considered as received after the application read or take 's1' or any other sample sent by the same *DataWriter* with a sequence number equal or greater than 'sn1'. For example:

```
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 these sample will not be
received again */
```

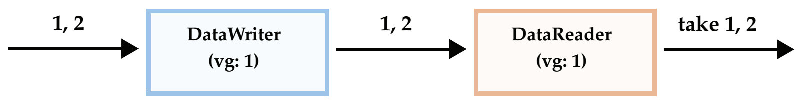
Important: For *DataReaders* on keyed topics requesting topic ordering, if you use read/take methods based on the instance handle, view_state, or instance_state, samples that have not been taken or read by the application yet may still be considered as received. To guarantee that every sample that is marked as received by the middleware has been read/taken by the application, you must use read/take methods that do not depend on the instance handle, view_state, or instance_state.

Important: The *DataReader* does not persist the full contents of the data in its historical cache; it only persists a record of which data it has delivered. 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 Read vs. Take (Section 7.4.3.1)) 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.

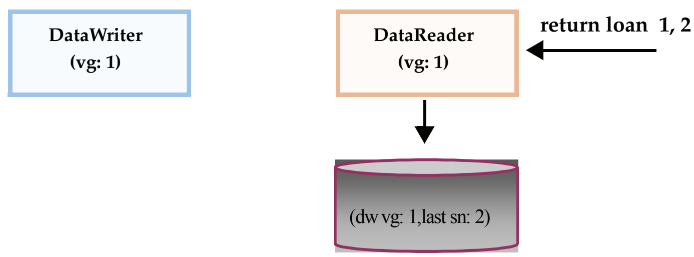
11.4.1 Durable Reader State Use Case

The following use case describes the durable reader state functionality:

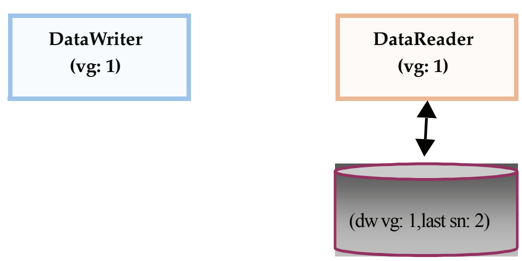
1. A *DataReader* receives two samples with sequence number 1 and 2 published by a *DataWriter* with virtual GUID 1. The application takes those samples.



2. After the application returns the loan on samples 1 and 2, the *DataReader* considers them as received 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 as received, the reader will not ask for these samples to the *DataWriter*.

11.4.2 How To Configure a DataReader for Durable Reader State

To configure a *DataReader* with durable reader state, use the PROPERTY QosPolicy (DDS Extension) (Section 6.5.15) 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 11.2 lists the supported properties.

Table 11.2 **Durable Reader State Properties**

Property	Description
dds.data_reader.state.odbc.dsn	**Required.** The ODBC DSN (Data Source Name) associated with the database where the *DataReader* state must be persisted.
dds.data_reader.state.filter_redundant_samples	To enable durable reader state, this property must be set to 1. When set to 0, the reader state is not maintained and *RTI Data Distribution Service* does not filter duplicate samples that may be coming from the same virtual writer. Default: 1
dds.data_reader.state.odbc.driver	This property indicates which ODBC driver to load. If the property is not specified, *RTI Data Distribution Service* will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).
dds.data_reader.state.odbc.username	These two properties configure the username and password used to connect to the database. Default: No password or username
dds.data_reader.state.odbc.password	
dds.data_reader.state.restore	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

Table 11.2 **Durable Reader State Properties**

Property	Description
dds.data_reader.state.checkpoint_frequency	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 *RTI Data Distribution Service* and they will not be propagated to the application. Default: 1
dds.data_reader.state.persistence_service.request_depth	This property indicates how many of the most recent historical samples the persisted *DataReader* wants to receive upon start-up. Default: 0

Example (C++ code):

```
/* 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 */
}

retcode = DDSPropertyQosPolicyHelper::add_property(
              readerQos.property,
              "dds.data_reader.state.restore",
              "<0|1>",
              DDS_BOOLEAN_FALSE);
```

```
if (retcode != DDS_RETCODE_OK) {
       /* Report error */
}
/* Create Data Reader */
...
```

11.5 Data Durability

The data durability feature is an implementation of the OMG DDS Persistence Profile. The DURABILITY QosPolicy (Section 6.5.6) allows an application to configure a *DataWriter* so that the information written by the *DataWriter* survives beyond the lifetime of the *DataWriter*.

RTI Data Distribution Service implements TRANSIENT and PERSISTENT durability using an external service called *RTI Persistence Service*, available for purchase as a separate RTI product.

RTI 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 DURABILITY QosPolicy (Section 6.5.6) to one of the following values:

- ❏ DDS_TRANSIENT_DURABILITY_QOS: *RTI Data Distribution Service* will store previously published samples in memory using *RTI Persistence Service*, which will send the stored data to newly discovered *DataReaders*.
- ❏ DDS_PERSISTENT_DURABILITY_QOS: *RTI Data Distribution Service* will store previously published samples in permanent storage, like a disk, using *RTI 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 DURABILITY QosPolicy (Section 6.5.6). A *DataReader* requesting PERSISTENT data will not receive data from *DataWriters* or *RTI Persistence Service* applications that are configured with TRANSIENT durability.

11.5.1 RTI Persistence Service

RTI Persistence Service is an *RTI Data Distribution Service* application that is configured to persist topic data. *RTI Persistence Service* is included with *RTI Data Distribution Service Professional Edition* and *Elite Edition.* 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-joiners *DataReaders*.

For more information on *RTI Persistence Service*, please see:

- ❏ Chapter 21: Introduction to RTI Persistence Service
- ❏ Chapter 22: Configuring RTI Persistence Service
- ❏ Chapter 23: Running RTI Persistence Service

RTI 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 *RTI 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 *RTI Persistence Service* (unless you are running multiple copies).
- ❏ **PERSISTENT mode** The PRSTDataWriters and PRSTDataReaders will be created with PERSISTENT durability and *RTI 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 *RTI Persistence Service*.

Peer-to-Peer Communication:

By default, a PERSISTENT/TRANSIENT *DataReader* 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 *RTI Persistence Service*. Duplicates will be discarded at the middleware level. This Peer-To-Peer communication pattern is illustrated inFigure 11.6. To use this peer-to-peer communication pattern, set the **direct_communication** field in the DURABILITY QosPolicy (Section 6.5.6) to TRUE. A PERSISTENT/TRANSIENT *DataReader* will receive information directly from PERSISTENT/TRANSIENT *DataWriters*.

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 *RTI Persistence Service*. This 'relay communication' pattern is illustrated in Figure 11.7. To use relay communication, set the **direct_communication** field in the DURABILITY QosPolicy (Section 6.5.6) to FALSE. A PERSISTENT/TRANSIENT *DataReader* will receive all the information from *RTI Persistence Service*.

Figure 11.6 **Peer-to-Peer Communication**

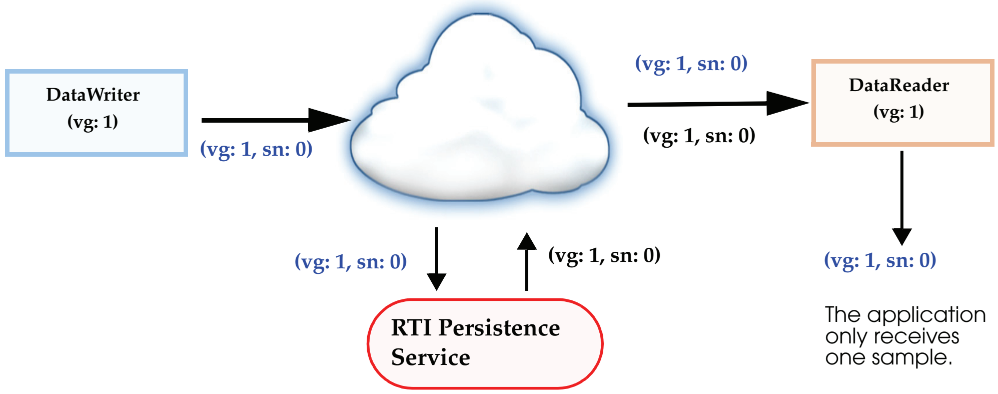


Figure 11.7 **Relay Communication**

Chapter 12 Discovery

This chapter discusses how *RTI Data Distribution Service* 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 *RTI Data Distribution Service* on the sending and receiving sides.

This chapter includes the following sections:

❏ What is Discovery? (Section 12.1)

❏ Configuring the Peers List Used in Discovery (Section 12.2)

❏ Discovery Implementation (Section 12.3)

❏ Debugging Discovery (Section 12.4)

❏ Ports Used for Discovery (Section 12.5)

The discovery process occurs automatically, so you do not have to implement any special code. We recommend that all users read What is Discovery? (Section 12.1) and Configuring the Peers List Used in Discovery (Section 12.2). 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 Chapter 13: Transport Plugins , as well as learning about these QosPolicies:

❏ TRANSPORT_SELECTION QosPolicy (DDS Extension) (Section 6.5.20)

❏ TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)

❏ TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21)

❏ TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5)

12.1 What is Discovery?

Discovery is the behind-the-scenes way in which *RTI Data Distribution Service* 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 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: Simple Participant Discovery (Section 12.1.1) and Simple Endpoint Discovery (Section 12.1.2). (Discovery can also be performed using the Enterprise Discovery Protocol—this requires a separately purchased package, *RTI Enterprise Discovery Service*.)

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 DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2). 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.

12.1.1 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 domain by sending participant declaration messages, also known *participant DATA* submessages. 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 USER_DATA QosPolicy (Section 6.5.23)) need to be propagated.

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. The GUID prefix is calculated from the IP address and the process ID. These values are stored in the *DomainParticipant's* WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9). The entityID is set by *RTI Data Distribution Service* (you may be able to change it in a future version).

Once a pair of remote participants have discovered each other, they can move on to the Endpoint Discovery phase, which is how *DataWriters* and *DataReaders* find each other.

12.1.2 Simple Endpoint Discovery

This phase of the Simple Discovery Protocol is performed by the Simple Endpoint Discovery Protocol (SEDP).

During the Endpoint Discovery phase, *RTI Data Distribution Service* 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 Section 12.3, 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 liveliness 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, *RTI Data Distribution Service* 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 QosPolicies (which includes having the same partition name string, see Section 6.4.5). Furthermore, if the *DomainParticipant* has been set up to ignore certain *DataWriters*/*DataReaders*, those entities will not be considered during the matching process. See Section 14.4.2 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.

Section 12.3 describes the details about the discovery process.

12.2 Configuring the Peers List Used in Discovery

The *RTI Data Distribution Service* discovery process will try to contact all possible participants on each remote node in the 'initial peers list,' which comes from the *initial_peers* field of the *DomainParticipant's* DISCOVERY QosPolicy.

The 'initial peers list' is just that: an *initial* list of peers to contact. Furthermore, the peers list merely contains *potential* peers—there is no requirement that there actually be *RTI Data Distribution Service* applications on the hosts in the list.

After startup, you can add to the 'peers list' with the **add_peer()** operation (see Adding to the Peers List (Section 8.5.2.3)). The 'peer list' may also grow as peers are automatically discovered (if **accept_unknown_peers** is TRUE, see Controlling Acceptance of Unknown Peers (Section 8.5.2.6)).

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 Section 12.2.3. The file must be in the same directory as your application's executable.
- ❏ An environment variable named **NDDS_DISCOVERY_PEERS**, defined as a comma-separated list of peer descriptors (see Section 12.2.2).

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, *RTI Data Distribution Service* will only listen for discovery traffic on the first multicast address (element 0) in **multicast_receive_addresses**.

If both the file and environment variable are found, the file takes precedence and the environment variable will be ignored.[1] If neither are found[2], *RTI Data Distribution Service* will use a default value. See the online documentation for details (in the section on the DISCOVERY QosPolicy).

The file and environment variable make it easy to reconfigure which nodes will take part in the discovery process—without recompiling your application.

The file and environment variable are the possible sources for the *default* initial peers list. You can, of course, explicitly set the initial list by calling the *DomainParticipant's* **set_default_qos()** operation, or add to the list after startup with the *DomainParticipant's* **add_peer()** operation (see Section 8.5.2.3).

If you set NDDS_DISCOVERY_PEERS *and* You Want to Communicate over Shared Memory:

Suppose you want to communicate with other *RTI Data Distribution Service* 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 Platform Notes document.)

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 TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8) 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.

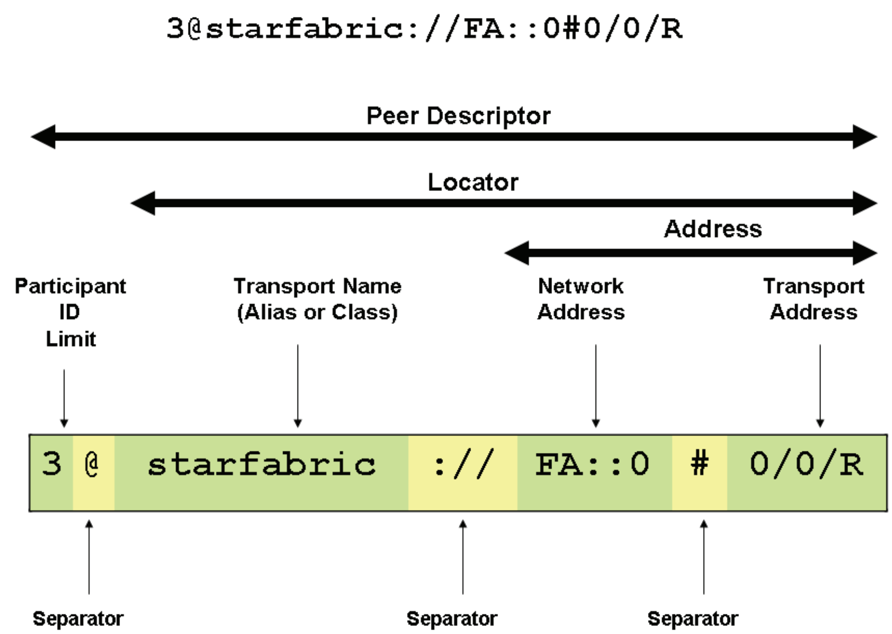
2. **Note for VxWorks 5.4 Users:** *RTI Data Distribution Service* will try to open the file with fopen(). If the NDDS_DISCOVERY_PEERS file is not accessible, on VxWorks 5.4 systems, you may see a file-system dependent error message. For example, when using the default 'rsh' file-system, you may see the following message: <hostName>:NDDS_DISCOVERY_PEERS: No such file or directory.

12.2.1 Peer Descriptor Format

A peer descriptor string specifies a range of participants at a given locator. Peer descriptor strings are used in the DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2) *initial_peers* field (see Section 8.5.2.2) and the *DomainParticipant's* **add_peer()** operation (see Section 8.5.2.3).

The anatomy of a peer descriptor is illustrated in Figure 12.1 using a special "StarFabric" transport example.

Figure 12.1 **Peer Descriptor Address String**



A peer descriptor consists of:

- ❑ *[optional]* A participant ID limit, which specifies the maximum participant ID that will be contacted by the *RTI Data Distribution Service* discovery process at the given locator. If omitted, a default value of 4 is implied.
- ❑ A locator, as described in Section 12.2.1.1.

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).

12.2.1.1 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 Section 12.2.1.2.

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

12.2.1.2 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 DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2) **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.

These are separated by the '#' character. If a separator is specified, it must be followed by a non-empty string which is passed to the transport plug-in.

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.

If you omit the '#' separator and the string is not a valid IPv4 or IPv6 address, it is treated as a transport address with an implicit network address (of the transport plug-in).

12.2.2 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 12.1 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 the Getting Started Guide).

Table 12.1 **NDDS_DISCOVERY_PEERS Environment Variable Examples**

NDDS_DISCOVERY_PEERS	Description of Host(s)
`239.255.0.1`	multicast
`localhost`	localhost
`192.168.1.1`	10.10.30.232 (IPv4)
`FAA0::1`	FAA0::0 (IPv6)
`himalaya,gangotri`	himalaya and gangotri
`1@himalaya,1@gangotri`	himalaya and gangotri (with a maximum participant ID of 1 on each host)
`FAA0::0localhost`	FAA0::0localhost (could be a UDPv4 transport plug-in registered at network address of FAA0::0) (IPv6)

Table 12.1 **NDDS_DISCOVERY_PEERS Environment Variable Examples**

NDDS_DISCOVERY_PEERS	Description of Host(s)
`udpv4://himalaya`	himalaya accessed using the "udpv4" transport plug-ins) (IPv4)
`udpv4://FAA0::0localhost`	localhost using the "udpv4" transport plug-ins) registered at network address FAA0::0
`udpv4://`	all unicast addresses accessed via the "udpv4" (UDPv4) transport plug-ins)
`0/0/R` `#0/0/R`	0/0/R (StarFabric)
`starfabric://0/0/R` `starfabric://#0/0/R`	0/0/R (StarFabric) using the "starfabric" (StarFabric) transport plug-ins
`starfabric://FBB0::0#0/0/R`	0/0/R (StarFabric) using the "starfabric" (StarFabric) transport plug-ins registered at network address FAA0::0
`starfabric://`	all unicast addresses accessed via the "starfabric" (StarFabric) transport plug-ins
`shmem://`	all unicast addresses accessed via the "shmem" (shared memory) transport plug-ins
`shmem://FCC0::0`	all unicast addresses accessed via the "shmem" (shared memory) transport plug-ins registered at network address FCC0::0

12.2.3 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 the 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 - Default 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 'updv4' 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)
```

12.3 Discovery Implementation

Note: this section contains advanced material not required by most users.

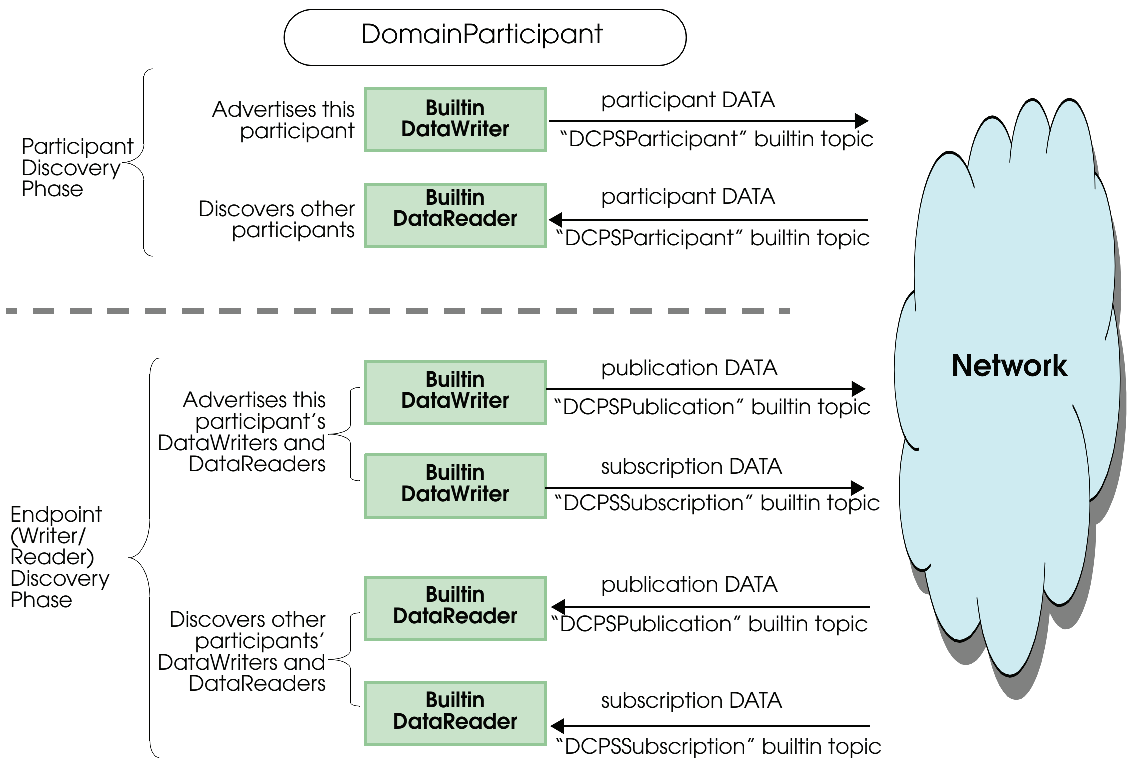
Discovery is implemented using built-in *DataWriters* and *DataReaders*. These are the same class of entities your application uses to send/receive data. That is, they are also of type **DDSDataWriter/DDSDataReader**. For each *DomainParticipant*, three built-in *DataWriters* and three *built-in DataReaders* are automatically created for discovery purposes. Figure 12.2 shows how these objects are used. (For more on built-in *DataReaders* and *DataWriters*, see Chapter 14: Built-In Topics).

The implementation is split into two separate protocols:

Simple Participant Discovery Protocol (SPDP)
\+ Simple Endpoint Discovery Protocol (SEDP)

= Simple Discovery Protocol (SDP)

Figure 12.2 **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.*

12.3.1 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 *DataReader*s are "special" because the *DataWriter* can send to a given list of destinations, regardless of whether there is an *RTI Data Distribution Service* 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 DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2). 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 USER_DATA QosPolicy (Section 6.5.23)), a new *participant DATA* will be sent immediately.

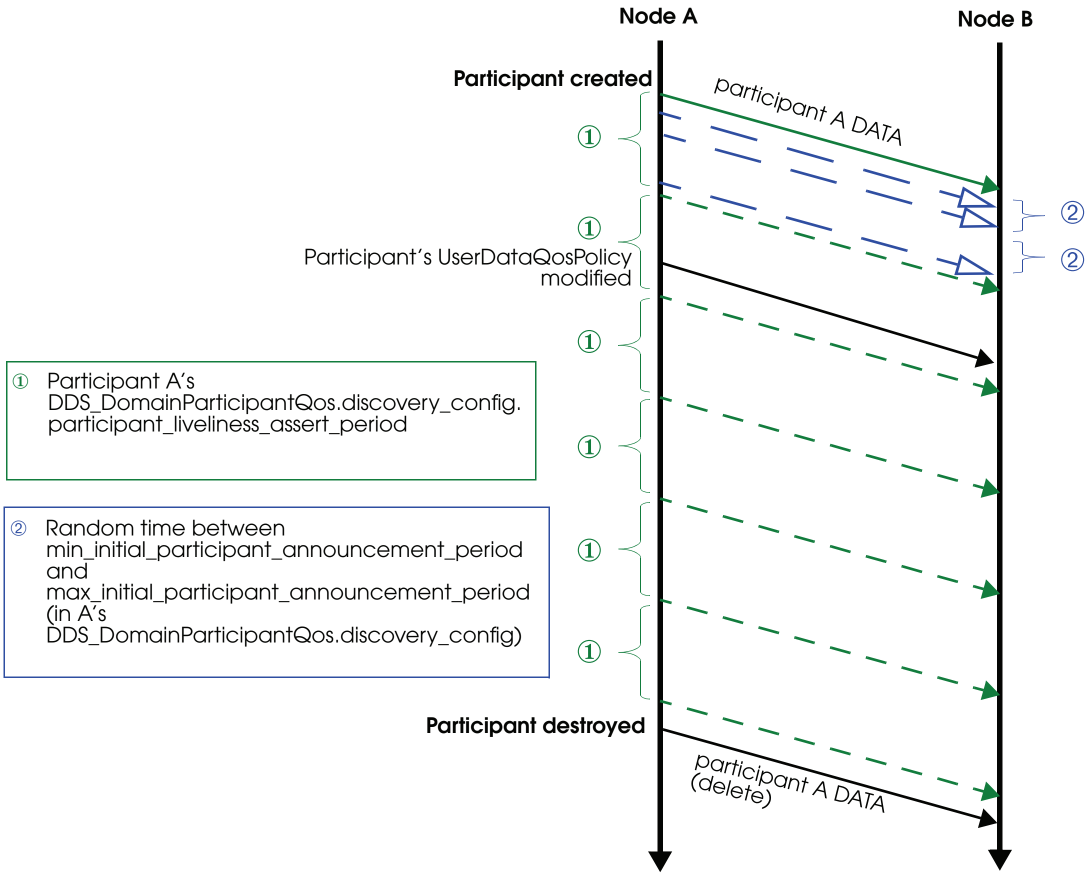
P*articipant DATAs* are also used to maintain a participant's liveliness status. These are sent at the rate set in the **participant_liveliness_assert_period** in the DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3).

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 DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2) is set to **TRUE**.

Once a remote participant has been added to the *RTI Data Distribution Service* database, *RTI Data Distribution Service* 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 liveliness. The rate at which the *participant DATA* is sent is controlled by the **participant_liveliness_assert_period** in the participant's DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3). This exchange, which keeps Participant A from appearing 'stale,' is illustrated in Figure 12.3. Figure 12.4 shows what happens when Participant A terminates ungracefully and therefore needs to be seen as 'stale.'

Figure 12.3 **Periodic 'participant DATAs'**

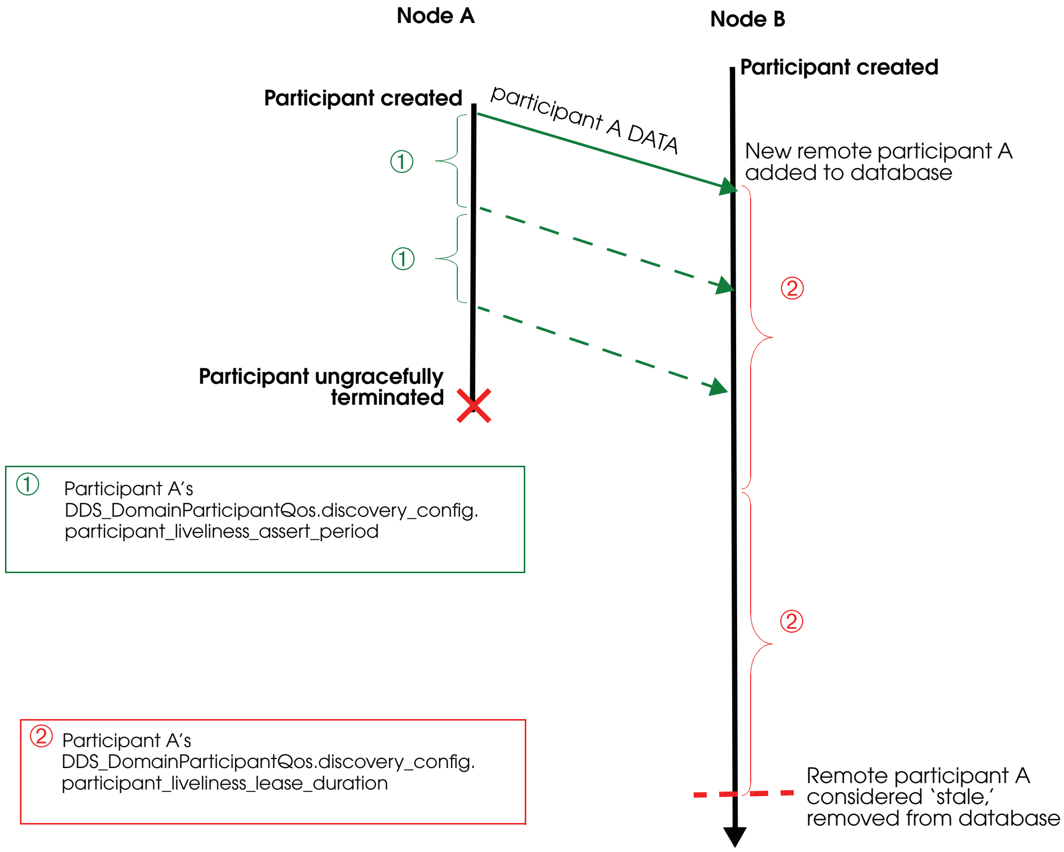


The DomainParticipant on Node A sends a 'participant DATA' to Node B, which is in Node A's peers list. This occurs regardless of whether or not there is an RTI Data Distribution Service application on Node B.

① *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.*

② *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.*

Figure 12.4 **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 12.3**.)*

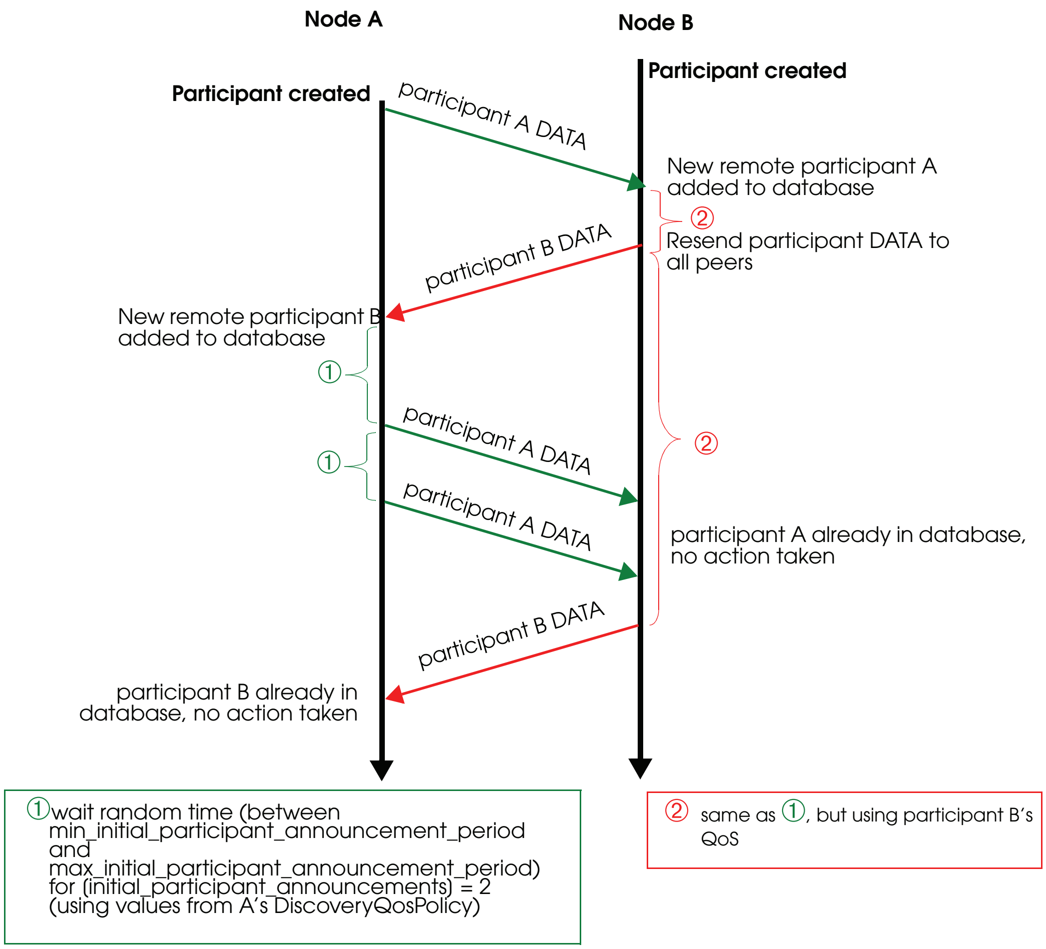
12.3.1.1 Refresh Mechanism

To ensure that a late-joining participant does not need to wait until the next refresh of the remote *participant DATA* to discover the remote participant, there is a resend mechanism. If the received *participant 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 DATA* to *all its peers*. This resend can potentially be done multiple times, with a random sleep time in between. Figure 12.5 illustrates this scenario.

The number of retries and the random amount of sleep between them are controlled by each participant's DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3) (see ① and ✍ in Figure 12.5).

Figure 12.6 provides a summary of the messages sent during the participant discovery phase.

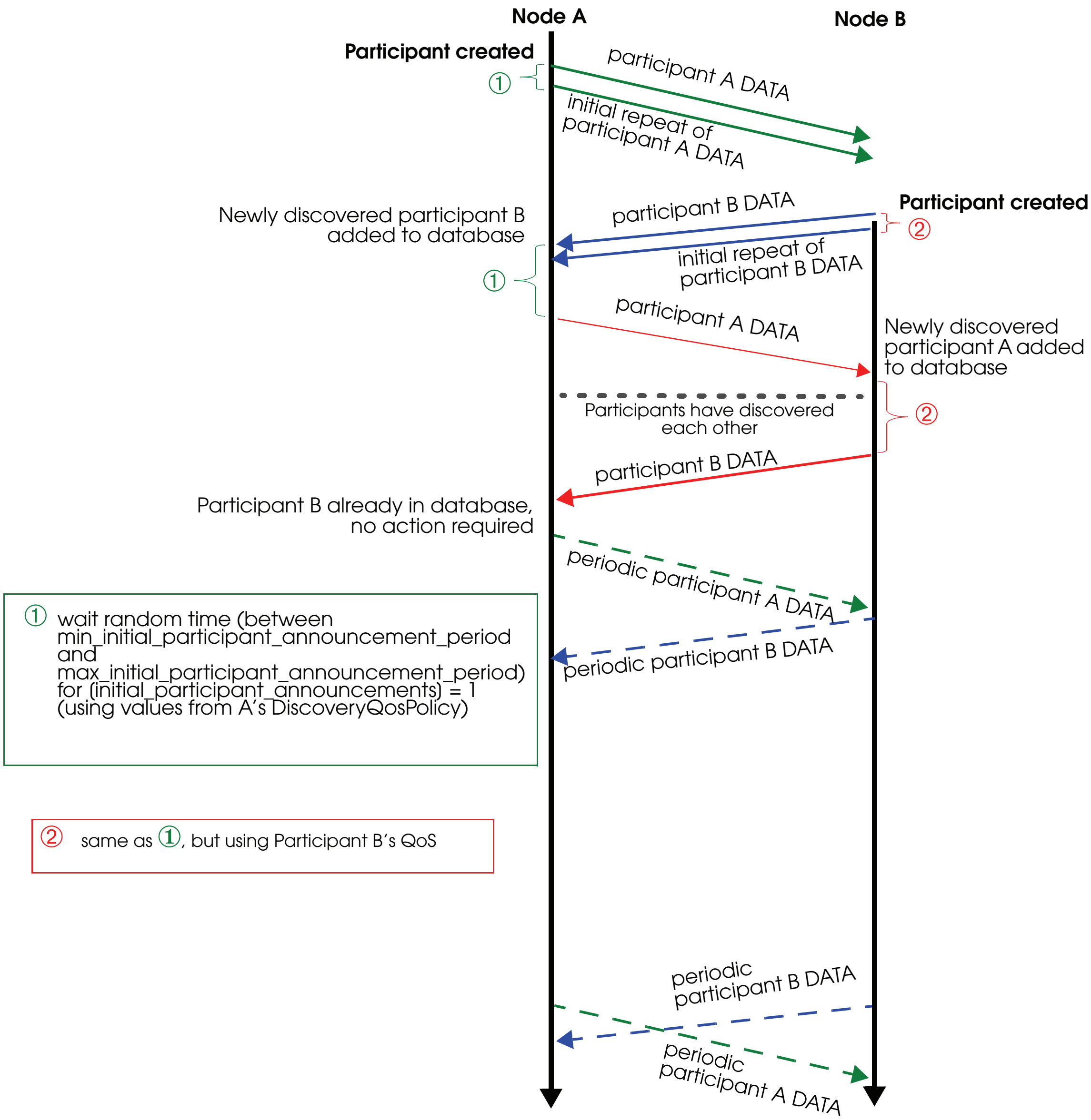
Figure 12.5 **Resending 'participant DATA' to a Late-Joiner**



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 ②, then resends its participant DATA.

(Initial repeat messages are omitted from this diagram for simplicity, see Figure 12.3*.)*

Figure 12.6 **Participant Discovery Summary**



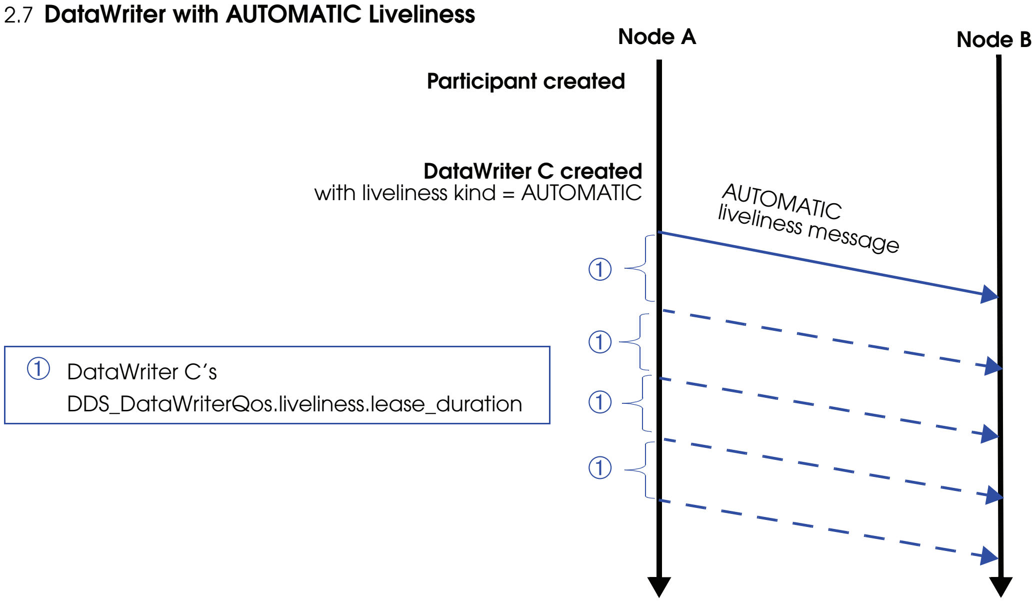
Participants A and B both have each other in their peers lists. Participant A is created first.

12.3.1.2 Maintaining DataWriter Liveliness for kinds AUTOMATIC and MANUAL_BY_PARTICIPANT

To maintain the liveliness of *DataWriters* that have a LIVELINESS QosPolicy (Section 6.5.11) **kind** field set to **AUTOMATIC** or **MANUAL_BY_PARTICIPANT**, *RTI Data Distribution Service* 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 the shortest **lease_duration** of these *DataWriters*. (The **lease_duration** is a field in the LIVELINESS QosPolicy (Section 6.5.11).) Figure 12.7 illustrates this scenario.

Figure 12.7 **DataWriter with AUTOMATIC Liveliness**



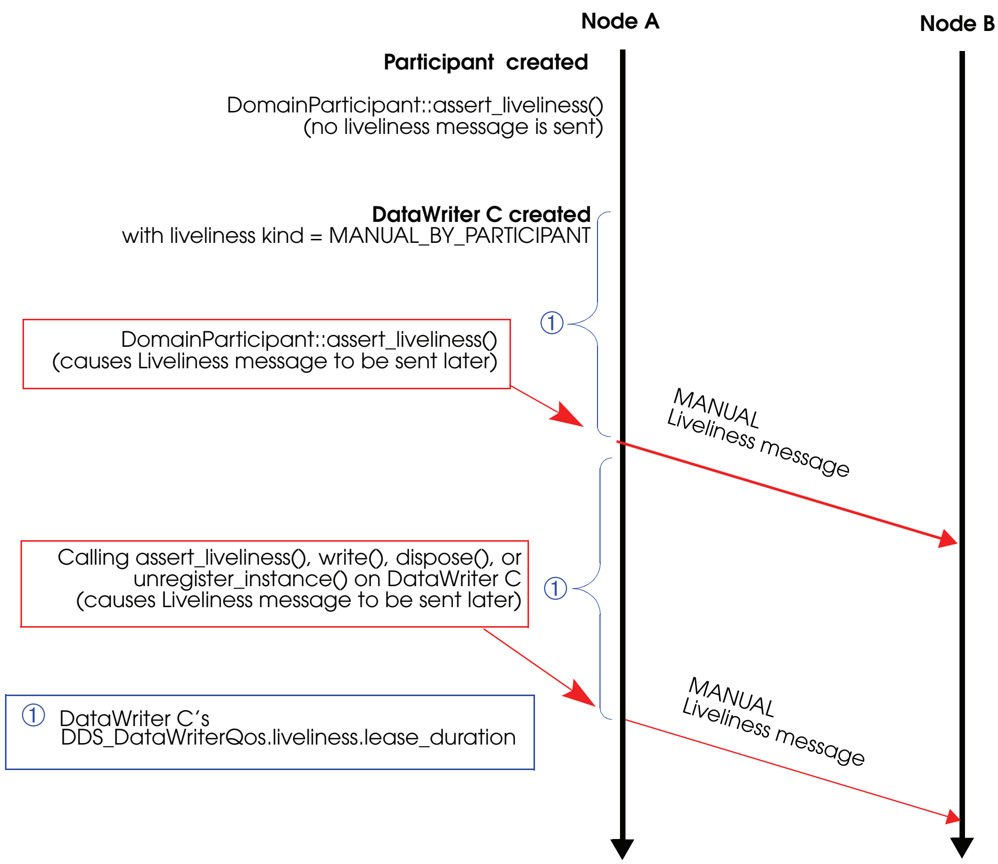
A liveliness message is sent automatically when a DataWriter with AUTOMATIC Liveliness kind is created, and then periodically, every DDS_DataWriterQos.liveliness.lease_duration.

If the *DomainParticipant* has any *DataWriters* with Liveliness QosPolicy **kind** set to **MANUAL_BY_PARTICIPANT**, *RTI Data Distribution Service* will periodically check to see if any of them have called **write()**, **assert_liveliness()**, **dispose()** or **unregister()**. The rate of this check is every X seconds, where X is the smallest **lease_duration** among all the *DomainParticipant's* **MANUAL_BY_PARTICIPANT** *DataWriters.* (The **lease_duration** is a field in the LIVELINESS QosPolicy (Section 6.5.11).) 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 time period. These **MANUAL** liveliness messages are used to update the liveliness of all the *DomainParticipant's* **MANUAL_BY_PARTICIPANT** *DataWriters*, as well as the liveliness of the *DomainParticipant* itself. Figure 12.8 shows an example sequence.

Figure 12.8 **DataWriter with MANUAL_BY_PARTICIPANT Liveliness**



Once a MANUAL_BY_PARTICIPANT 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, and the participant itself.

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 LIVELINESS QosPolicy (Section 6.5.11) **kind** set to **AUTOMATIC** or **MANUAL_BY_PARTICIPANT**, then no liveliness messages are ever sent from the inter-participant writer.

12.3.2 Endpoint Discovery

As we saw in Figure 12.2 on page 12-11, 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, *RTI Data Distribution Service* assumes that the remote *DomainParticipant* is implemented in the same way and therefore is creating the appropriate counterpart entities. Therefore, *RTI Data Distribution Service* 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 *DomainParticipant*.

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 *DomainParticipant*s 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 *DomainParticipant*s:

❏ TOPIC_DATA QosPolicy (Section 5.2.1)

❏ GROUP_DATA QosPolicy (Section 6.4.4)

❏ USER_DATA QosPolicy (Section 6.5.23)

❏ DEADLINE QosPolicy (Section 6.5.4)

- ❏ OWNERSHIP_STRENGTH QosPolicy (Section 6.5.14)
- ❏ PARTITION QosPolicy (Section 6.4.5)
- ❏ TIME_BASED_FILTER QosPolicy (Section 7.6.4)
- ❏ LIFESPAN QoS Policy (Section 6.5.10)

What the above QosPolicies have in common is that they are all changeable and part of the built-in data (see Chapter 14: Built-In Topics).

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 DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3). 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 12.9.

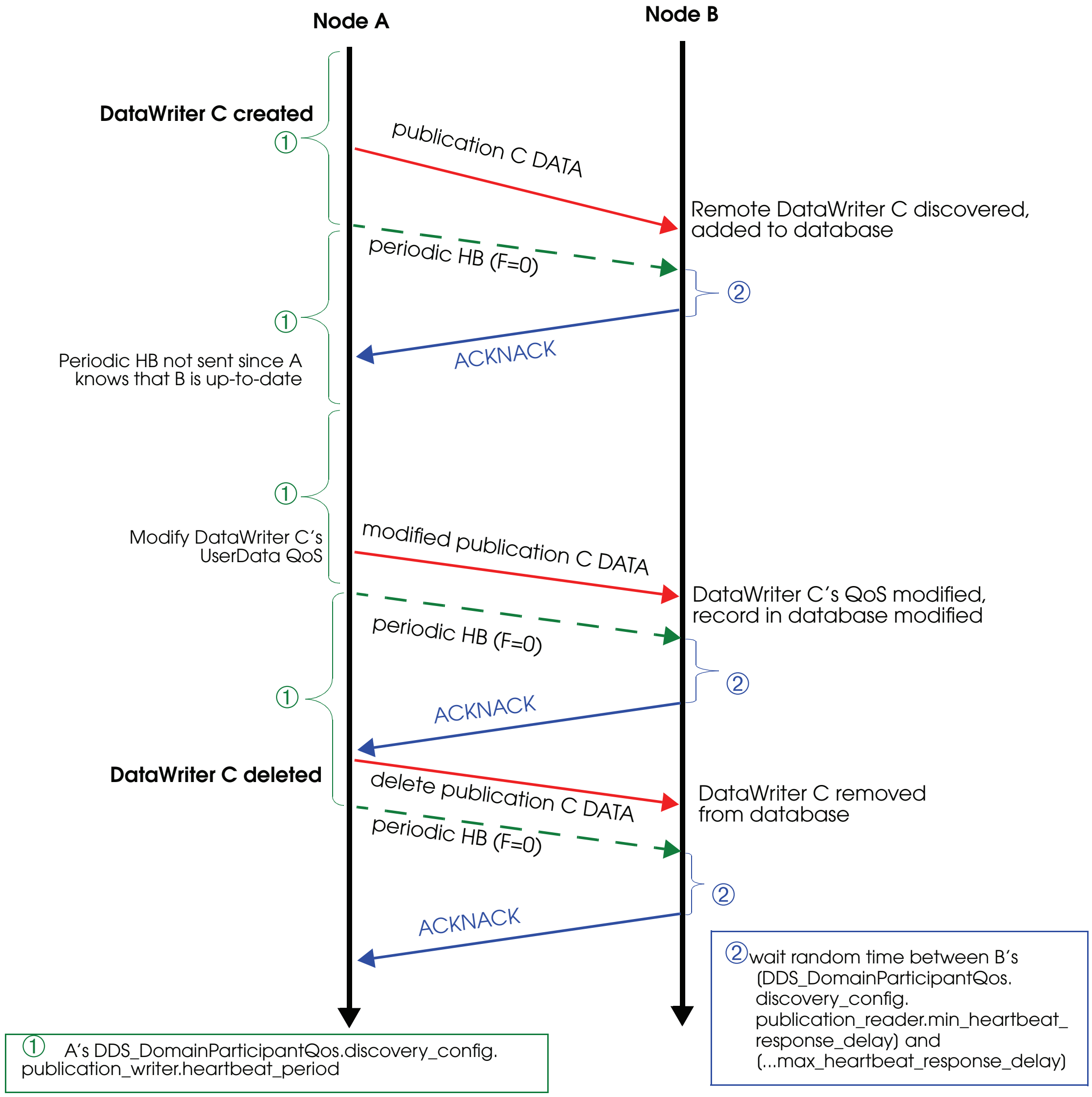
Discovery endpoint writers and readers have their HISTORY QosPolicy (Section 6.5.8) set to **KEEP_LAST**, and their DURABILITY QosPolicy (Section 6.5.6) set to **TRANSIENT_LOCAL**. Therefore, even if the remote *DomainParticipant* has not yet been discovered at the time the local user's *DataWriter/DataReader* 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 12.10 and Figure 12.11 illustrate this sequence for HB and ACK/NACK triggers, respectively.

Endpoint discovery latency is determined by the following members of the *DomainParticipant's* DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3):

- ❏ publication_writer
- ❏ subscription_writer
- ❏ publication_reader
- ❏ subscription_reader

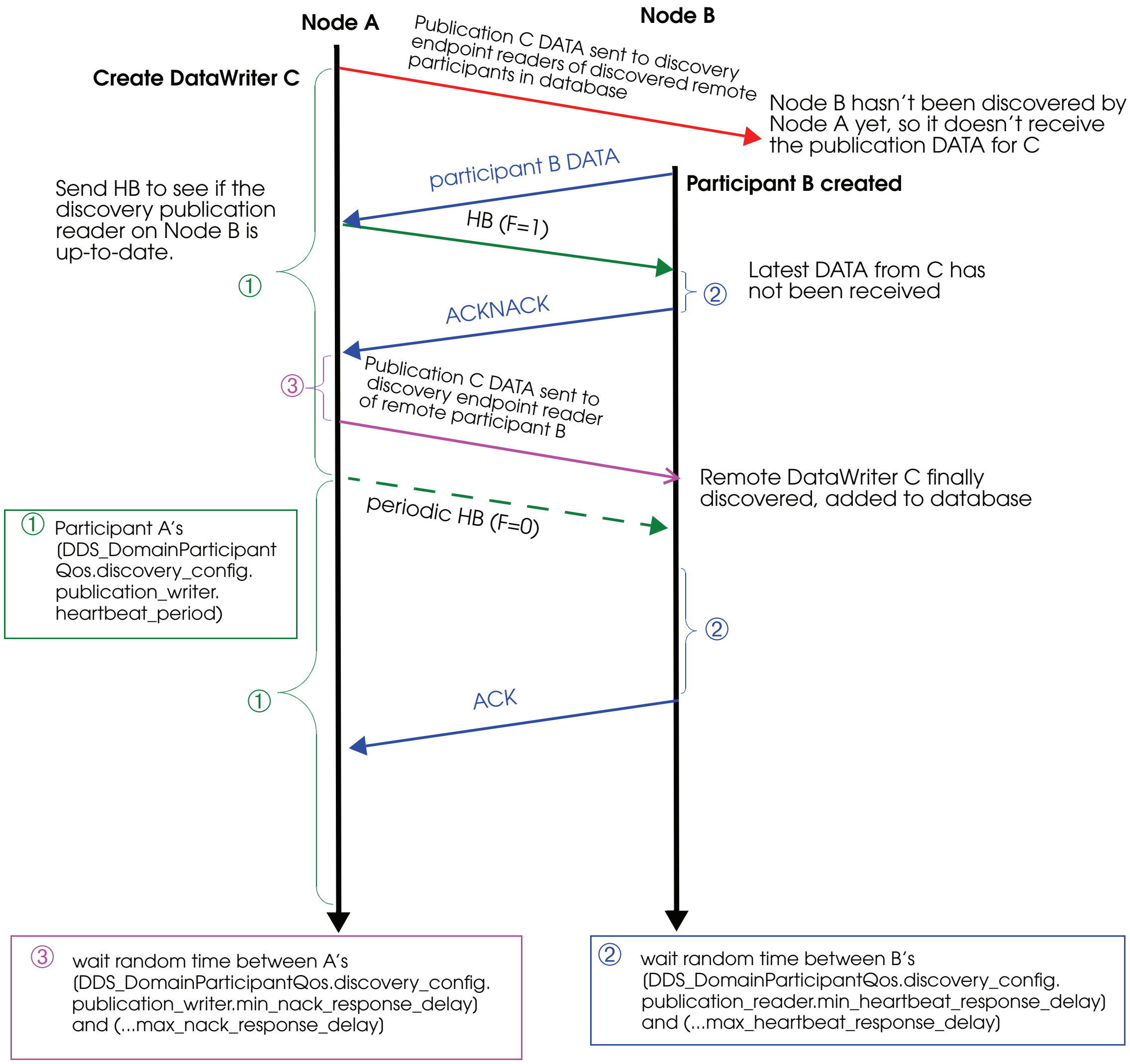
When a remote entity record is added, removed, or changed in the database, matching is performed with all the local entities. Only after there is a successful match on both ends can an application's user-created *DataReaders* and *DataWriters* communicate with each other.

Figure 12.9 **Endpoint Discovery Summary**



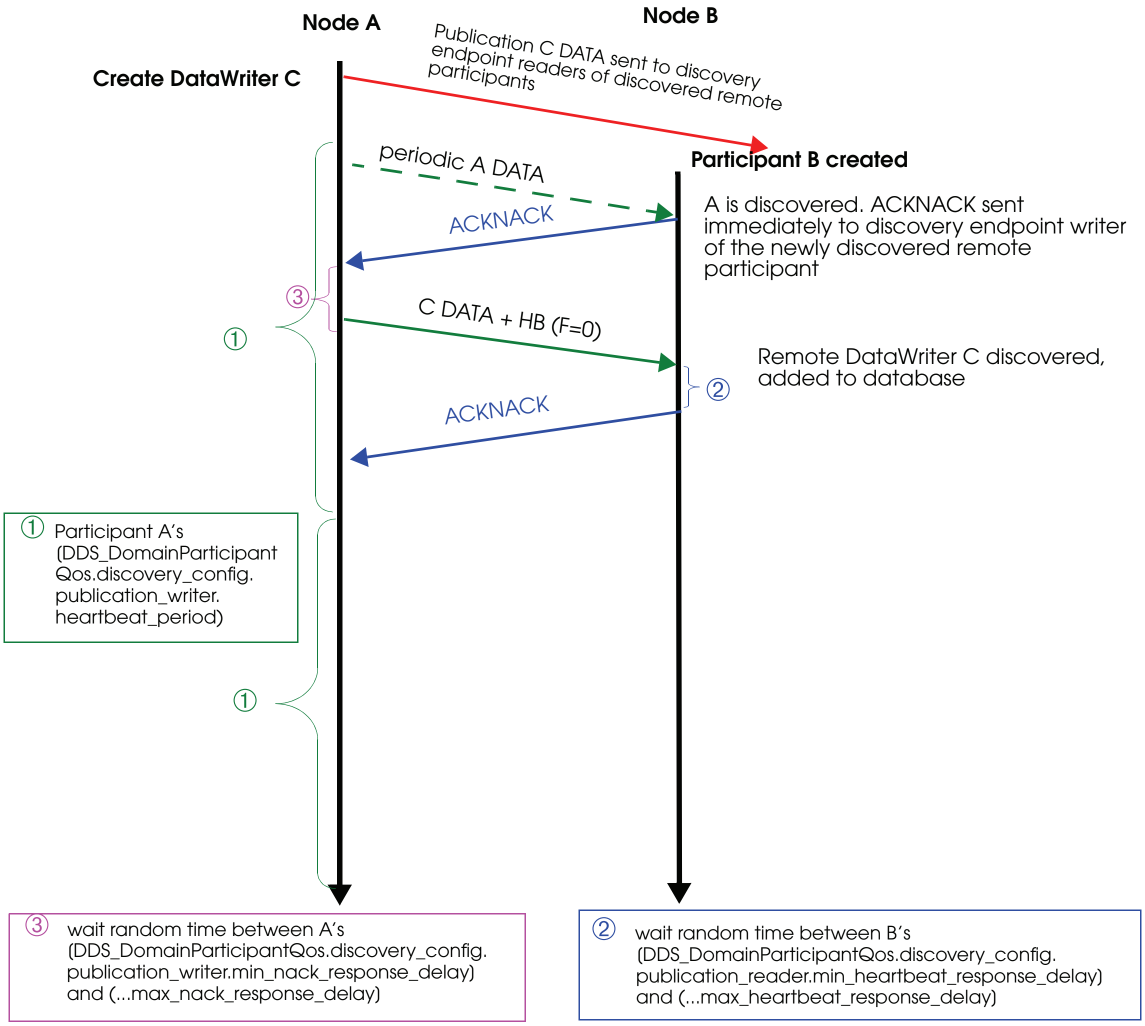
Assume participants A and B have been discovered on both sides. A's DiscoveryConfigQosPolicy.publication_writer.heartbeats_per_max_samples = 0, so no HB is piggybacked with the publication DATA. A HB with F=0 is a request for an ACK/NACK. The periodic and initial repeat participant DATAs are omitted from the diagram.

Figure 12.10 **DataWriter Discovered by Late-Joiner, Triggered by HB**



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 12.3.*)*

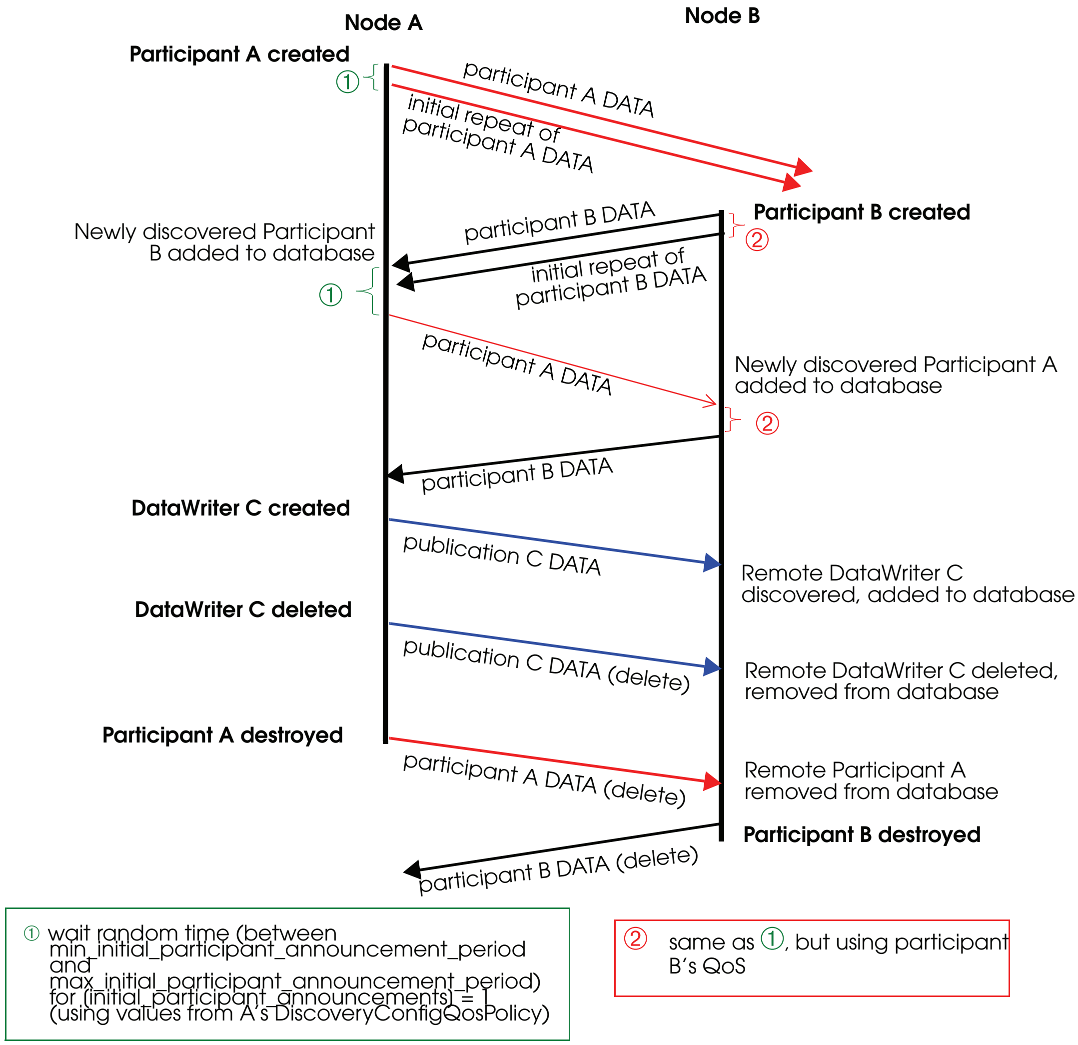
Figure 12.11 **DataWriter Discovered by Late-Joiner, Triggered by ACKNACK**



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 message. Participant A has B in its peer list, but not vice versa. Accept_unknown_locators is true. In response to receiving the new Participant A DATA message on node B, a participant B DATA message will be sent to A. The discovery endpoint writer on Node B will also send a HB to the discovery endpoint reader on Node A. These are omitted in the diagram for simplicity. (Initial repeat participant messages and periodic participant messages are omitted from this diagram, see Figure 12.3.*)*

For more information about reliable communication, see Chapter 10: Reliable Communications.

12.3.3 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.

12.3.4 Discovery-Related QoS

Each *DomainParticipant* needs to be uniquely identified in the domain and specify which other *DomainParticipant*s it is interested in communicating with. The WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9) uniquely identifies a *DomainParticipant* in the domain. The DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2) 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 DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)'s **participant_liveliness_assert_period** and **participant_liveliness_lease_duration** fields are set to small values, the discovery of stale remote *DomainParticipant*s will occur faster, but more discovery traffic will be sent over the network. Setting the participant's **heartbeat_period**[1] to a small value can cause late-joining *DomainParticipant*s to discover remote user-data *DataWriters* and *DataReaders* at a faster rate, but *RTI Data Distribution Service* might send HBs to other nodes more often. This timing can be controlled by the following *DomainParticipant* QosPolicies:

DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2) — specifies how other *DomainParticipant*s in the network can communicate with this *DomainParticipant*, and which other *DomainParticipant*s in the network this *DomainParticipant* is interested in communicating with. See also:Ports Used for Discovery (Section 12.5)

DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3) — specifies the QoS of the discovery readers and writers (parameters that control the HB and ACK rates of discovery endpoint readers/writers, and periodic refreshing of *participant DATA* from discovery participant readers/writers).

DOMAIN_PARTICIPANT_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 8.5.4) — specifies the number of local and remote entities expected in the system.

WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9) — specifies the **rtps_app_id** and **rtps_host_id** that uniquely identify the participant in the domain.

The other important parameter is the domain ID: *DomainParticipant*s can only discover each other if they belong to the same domain. The domain ID is a parameter passed to the **create_participant()** operation (see Section 8.3.1).

1. heartbeat_period is part of the DDS_RtpsReliableWriterProtocol_t structure used in the DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3)'s publication_writer and subscription_writer fields.

12.4 Debugging Discovery

To understand the flow of messages during discovery, you can increase the verbosity of the messages logged by *RTI Data Distribution Service* 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:

```
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 Section 12.3.3, these are the messages as seen on DomainParticipant A:

```
[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:announc-
ing new publication: 0XA0A01A1,0X5522,0X1,0X80000003
DISCSimpleEndpointDiscoveryPluginPDFListener_onAfterLocalWriterEnabled:at
{46c614d9,101615EB}
```

- (The above messages mean: Publication C DATA has been sent.)

```
DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAvailable:dis-
covered subscription: 0XA0A01A1,0X552B,0X1,0X80000004
DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAvailable:at
{46c614dd,94FAEFEF}
DISCEndpointDiscoveryPlugin_assertRemoteEndpoint:plugin discovered/updated
remote endpoint: 0XA0A01A1,0X552B,0X1,0X80000004
DISCEndpointDiscoveryPlugin_assertRemoteEndpoint:at {46c614dd,950203DF}
```

- (The above messages mean: Receiving subscription D DATA from Node B.)

```
PRESPsService_linkToLocalWriter:assert remote 0XA0A01A1,0X552B,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:announc-
ing disposed local publication: 0XA0A01A1,0X5522,0X1,0X80000003
[D0049|DELETE_CONTAINED]DISCPluginManager_onAfterLocalEndpointDeleted:at
{46c61501,288051C8}
[D0049|DELETE_CONTAINED]DISCSimpleEndpointDiscoveryPluginPDFListener_onAfterLoca
lWriterDeleted:announcing disposed publication: 0XA0A01A1,0X5522,0X1,0X80000003
[D0049|DELETE_CONTAINED]DISCSimpleEndpointDiscoveryPluginPDFListener_onAfterLoca
lWriterDeleted: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.)

```
DISCParticipantDiscoveryPlugin_removeRemoteParticipantsByCookie:plugin removing
3 remote entities by cookie
DISCParticipantDiscoveryPlugin_removeRemoteParticipantsByCookie:at
{46c61501,28E38A7C}
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:plugin discovered dis-
posed remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:at {46c61501,28E68E3D}
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:remote entity removed
from database: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:at {46c61501,28E68E3D}
```

- (The above messages mean: Removing discovered entities from local database, before shutting down.)

As you can see, the messages are encoded, since they are primarily used by RTI support personnel.

For more information on the message logging API, see Controlling Messages from RTI Data Distribution Service (Section 18.2).

If you notice that a remote entity is not being discovered, check the QoS related to discovery (see Section 12.3.4).

If a remote entity is discovered, but does not match with a local entity as expected, check the QoS of both the remote and local entity.

12.5 Ports Used for Discovery

There are two kinds of traffic in an *RTI Data Distribution Service* 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, *RTI Data Distribution Service* uses different ports, as described below.

Note: The ports described in this section are used for *incoming* data. *RTI Data Distribution Service* uses ephemeral ports for outbound data.

RTI Data Distribution Service 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 *RTI Data Distribution Service* to establish the presence of remote DDS 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 *RTI Data Distribution Service*. These parameters are shown in Table 12.2. (For defaults and valid ranges, please see the online documentation.)

Table 12.2 **WireProtocol QosPolicy's rtps_well_known_ports (DDS_RtpsWellKnownPorts_t)**

<table>
<tr><th>Type</th><th>Field Name</th><th>Description</th></tr>
<tr><td rowspan="7">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>domain_id_gain</td><td rowspan="2">Tunable gain parameters. See Section 12.5.4.</td></tr>
<tr><td>participant_id_gain</td></tr>
<tr><td>builtin_multicast_port_offset</td><td rowspan="2">Additional offset for meta-traffic port. See Inbound Ports for Meta-Traffic (Section 12.5.1).</td></tr>
<tr><td>builtin_unicast_port_offset</td></tr>
<tr><td>user_multicast_port_offset</td><td rowspan="2">Additional offset for user traffic port. See Inbound Ports for User Traffic (Section 12.5.2).</td></tr>
<tr><td>user_unicast_port_offset</td></tr>
</table>

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 12.2, the port formulas described below depend on:

- ❏ The Domain ID specified when the *DomainParticipant* is created (see Section 8.3.1). The Domain ID ensures no port conflicts exist between Participants belonging to different domains. This also means that discovery traffic in one domain is not visible to *DomainParticipant*s in other domains.
- ❏ The **participant_id** is a field in the WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9), see Section 8.5.9.1. The **participant_id** ensures that unique unicast port numbers are assigned to *DomainParticipant*s belonging to the same domain on a given host.

Backwards Compatibility: *RTI Data Distribution Service* 4.5 supports the standard DDS Interoperability Wire Protocol based on the Real-time Publish-Subscribe (RTPS) protocol. Because this protocol is not compatible with that used by prior versions of *RTI Data Distribution Service* (4.2c and older), applications built with 4.5 (as well as 4.4, 4.3, and 4.2e) will not interoperate with applications built with versions 4.2c and older. 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 domains in the system and the maximum number of participants per 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].

12.5.1 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:

```
metatraffic_unicast_port = port_base +
                           (domain_id_gain * Domain ID) +
                           (participant_id_gain * participant_id) +
                            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 Section 8.5.2.4).

```
metatraffic_multicast_port = port_base +
                             (domain_id_gain * Domain ID) +
                              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 DISCOVERY_CONFIG QosPolicy (DDS Extension) (Section 8.5.3) is set.

12.5.2 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* TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5) and TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21), or the *DataWriter's* TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21).

The WireProtocol QosPolicy's **rtps_well_known_ports.usertraffic_unicast_port** determines the port used for receiving user data using unicast:

```
usertraffic_unicast_port = port_base +
                           (domain_id_gain * Domain ID) +
                           (participant_id_gain * participant_id)+
                            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 TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21).

```
usertraffic_multicast_port = port_base +
                             (domain_id_gain * Domain ID) +
                              user_multicast_port_offset
```

12.5.3 Automatic Selection of participant_id and Port Reservation

The WIRE_PROTOCOL QosPolicy (DDS Extension) (Section 8.5.9) **rtps_reserved_ports_mask** field determines what type of ports are reserved when the *DomainParticipant* is enabled. See Choosing Participant IDs (Section 8.5.9.1).

12.5.4 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** (Section 12.5.4), 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 domain occupy a consecutive range of **domain_id_gain** ports. Precisely, all ports occupied by the domain fall within:

  ```
  (port_base + (domain_id_gain * Domain ID))
  ```

  and:

  ```
  (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:

  ```
  max_participant_id < (domain_id_gain / participant_id_gain)
  ```

- On the contrary, if **domain_id_gain** <= **participant_id_gain**, it results in a port mapping layout where a given 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>
<tr><th>Mapped Port</th><th>Domain ID</th><th>Participant ID</th></tr>
<tr><td rowspan="5">higher port number</td><td>1</td><td rowspan="2">2</td></tr>
<tr><td>0</td></tr>
<tr><td>1</td><td rowspan="2">1</td></tr>
<tr><td>0</td></tr>
<tr><td>1</td><td rowspan="2">0</td></tr>
<tr><td>lower port number</td><td>0</td></tr>
</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:

  ```
  max_domain_id < (participant_id_gain / domain_id_gain)
  ```

The **domain_id_gain** also determines the range of the port-specific offsets:

```
domain_id_gain >
abs(builtin_multicast_port_offset - user_multicast_port_offset)
```

and

```
domain_id_gain >
abs(builtin_unicast_port_offset - 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**.

```
participant_id_gain > abs(builtin_unicast_port_offset -
user_unicast_port_offset)
```

In all cases, the resulting ports must be within the range imposed by the underlying transport.

Chapter 13 Transport Plugins

RTI Data Distribution Service has a pluggable transports architecture. The core of *RTI Data Distribution Service* is transport agnostic—it does not make any assumptions about the actual transports used to send and receive messages. Instead, *RTI Data Distribution Service* 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** *RTI Data Distribution Service* comes with a set of commonly used transport plugins. These 'builtin' plugins include UDPv4, UDPv6, and shared memory. So that *RTI Data Distribution Service* applications can work out-of-the-box, some of these are enabled by default (see TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)).
- **Extension Transport Plugins** RTI offers extension transports, including *RTI Secure WAN Transport* (see Chapter 19 and Chapter 20) and *RTI TCP Transport* (see Chapter 29).
- **Custom-developed Transport Plugins** RTI supports the use of custom transport plugins. This is a powerful capability that distinguishes *RTI Data Distribution Service* from competing middleware approaches. If you are interested in developing a custom transport plugin for *RTI Data Distribution Service*, please contact your local RTI representative or email **sales@rti.com**.

This chapter describes the following:

- Builtin Transport Plugins (Section 13.1)
- Extension Transport Plugins (Section 13.2)
- The NDDSTransportSupport Class (Section 13.3)

- ❑ Explicitly Creating Builtin Transport Plugin Instances (Section 13.4)
- ❑ Setting Builtin Transport Properties of the Default Transport Instance—get/set_builtin_transport_properties() (Section 13.5)
- ❑ Setting Builtin Transport Properties with the PropertyQosPolicy (Section 13.6)
- ❑ Installing Additional Builtin Transport Plugins with register_transport() (Section 13.7)
- ❑ Installing Additional Builtin Transport Plugins with PropertyQosPolicy (Section 13.8)
- ❑ Other Transport Support Operations (Section 13.9)

13.1 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 TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8) 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*[1] the transports are registered.
- ❑ *Other Transport Instances*: There are two ways to install non-default builtin 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_Shmem_new()**, see Section 13.4), then calling **register_transport()** (Section 13.7). (For example, suppose you want an extra instance of a transport.) (Not available for the Java or .NET API.)
  - Additional builtin transport instances can also be installed through the PROPERTY QosPolicy (DDS Extension) (Section 6.5.15).

1. Any transport property changes made after the plugin is registered will have no effect.

To configure the properties of the builtin transports:

❑ Set properties by calling **set_builtin_transport_property()** (see Section 13.5)

or

❑ Specify predefined property strings in the *DomainParticipant's* PropertyQosPolicy, as described in Section 13.6.

❑ For other builtin transport instances:

- If the builtin transport plugin is created with **NDDS_Transport_UDPv4_new()**, **NDDS_Transport_UDPv6_new()** or **NDDS_Transport_Shmem_new()**, properties can be specified during creation time. See Explicitly Creating Builtin Transport Plugin Instances (Section 13.4).
- If the additional builtin transport instances are installed through the PROPERTY QosPolicy (DDS Extension) (Section 6.5.15), the properties of the builtin transport plugins can also be specified through that same QosPolicy.

13.2 Extension Transport Plugins

If you want to change the properties for an extension transport plugin, do so *before*[1] the plugin is registered.

There are two ways to install an extension transport plugin:

❑ *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.

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. For more information, see Chapter 15: Configuring QoS with XML.

1. Any transport property changes made after the plugin is registered will have no effect.

- *Explicit Registration*: Transports may be explicitly registered by first creating an instance of the transport plugin (see Section 13.4) and then calling **register_transport()** (see Section 13.7).

13.3 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 13.1.

Table 13.1 **Transport Support Operations**

Operation	Description	Reference
register_transport	Registers a transport plugin for use with a DomainParticipant.	Section 13.7
get_builtin_transport_property	Gets the properties used to create a builtin transport plugin.	Section 13.5
set_builtin_transport_property	Sets the properties used to create a builtin transport plugin.	
add_send_route	Adds a route for outgoing messages.	Section 13.9.1
add_receive_route	Adds a route for incoming messages.	Section 13.9.2
lookup_transport	Looks up a transport plugin within a DomainParticipant.	Section 13.9.3

13.4 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 TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)). 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()** (Section 13.7). (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_Shmem_new (
    const struct NDDS_Transport_Shmem_Property_t * property_in)
```

property_in Desired behavior of this transport. May be NULL for default properties.

For details on using these functions, please see the online (HTML) documentation.

Your application may create and register multiple instances of these transport plugins with *RTI Data Distribution Service*. This may be done to partition the network interfaces across multiple *RTI Data Distribution Service* 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.

13.5 Setting Builtin Transport Properties of the 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 online/HTML documentation.) Used together, the two operations below allow you to customize properties of the builtin transport when it is implicitly registered (see Section 13.1).

Note: Another way to change the properties is with the Property QosPolicy, see Section 13.6. 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,
     const struct NDDS_Transport_Property_t & builtin_transport_property_in)
```

participant_in A valid non-NULL *DomainParticipant* that has not been enabled. If the *DomainParticipant* if 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_inout (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.

builtin_transport_property_in (Used by the "set" operation only.) The new transport property that will be used to the 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 reaching 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 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: Builtin 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.

Note: If **message_size_max** is increased from the default for any of the built-in transports, or if custom transports are used, then the **buffer_size** in the RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7) of the *DomainParticipant* should also be increased accordingly.

13.6 Setting Builtin Transport Properties with the PropertyQosPolicy

The PROPERTY QosPolicy (DDS Extension) (Section 6.5.15) 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:

```
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 6.48, “PropertyQoSPolicyHelper Operations,” on page 6-139, as well as the online (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 13.2, “Properties for the Builtin UDPv4 Transport,” on page 13-8
- ❑ Table 13.3, “Properties for the Builtin UDPv6 Transport,” on page 13-15
- ❑ Table 13.4, “Properties for the Builtin Shared Memory Transport,” on page 13-21

See also:

- ❑ “Notes Regarding Loopback and Shared Memory” (Section 13.6.1 on page 13-25)
- ❑ “Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6” (Section 13.6.2 on page 13-25)

Note: Changing properties with the PROPERTY QosPolicy (DDS Extension) (Section 6.5.15) will overwrite any properties set by calling **set_builtin_transport_property()**.

Table 13.2 **Properties for the Builtin UDPv4 Transport**

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
parent.address_bit_count	Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, the address_bit_count should be set to 8. For the range of addresses used by IPv4 (4 bytes), it should be set to 32.
parent.properties_bitmap	A bitmap that defines various properties of the transport to the *RTI Data Distribution Service* core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* from trying to gather too many buffers into a send call for the transport plugin. *RTI Data Distribution Service* requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN. See Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6 (Section 13.6.2).

Table 13.2 **Properties for the Builtin UDPv4 Transport**

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	**Property Value Description**
parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plugin. This value must be set before the transport plugin is registered, so that *RTI Data Distribution Service* can properly use the plugin. If you set this higher than the default, then the *DomainParticipant's* ReceiverPoolQosPolicy's **buffer_size** should also be changed. **Note:** If two transports have different settings for message_size_max , such that fragmentation is required for one and not the other, the need for fragmentation may not be applied correctly and samples may be lost. To avoid this issue, <u>set message_size_max to the same value in each transport</u>.

parent.allow_interfaces_list	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.
	For example, the following are acceptable strings
	`192.168.1.1`
	`192.168.1.*`
	`192.168.*`
	`192.*`
	If the list is non-empty, this "white" 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. 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.

Table 13.2 **Properties for the Builtin UDPv4 Transport**

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
parent.deny_interfaces_list	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. For example, the following are acceptable strings: 192.168.1.1 192.168.1.* 192.168.* 192.* This "black" list is applied after the parent.allow_interfaces_list 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 *DomainParticipant* is deleted.
parent. allow_multicast_interfaces_list	A list of strings, each identifying a range of interface addresses. 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 parent.allow_interfaces_list "white" list *and* the parent.deny_interfaces_list "black" list. From that resulting list, parent. deny_multicast_interfaces_list 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.

Table 13.2 **Properties for the Builtin UDPv4 Transport**

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
parent.deny_multicast_interfaces_list	A list of strings, each identifying a range of interface addresses. 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 "black" list is applied after the parent.allow_multicast_interfaces_list 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 *DomainParticipant* is deleted.
send_socket_buffer_size	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. The maximum value is operating system-dependent. If set to NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT, then **setsockopt()** (or equivalent) will not be called to size the send buffer of the socket.
recv_socket_buffer_size	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. The maximum value is operating system-dependent. Default: NDDS_TRANSPORT_UDPV4_MESSAGE_SIZE_MAX_DEFAULT. If set to NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT, then **setsockopt()** (or equivalent) will not be called to size the receive buffer of the socket.

Table 13.2 **Properties for the Builtin UDPv4 Transport**

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
unicast_enabled	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).
multicast_enabled	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).
multicast_ttl	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.
multicast_loopback_disabled	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. This is set to 0 by default. So multicast loopback is enabled. Turning off multicast loopback (set to 1) may result in minor performance gains when using multicast. **Note:** Windows CE does not support multicast loopback. This field is ignored for Windows CE targets.

Table 13.2 **Properties for the Builtin UDPv4 Transport**

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
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. Lets *RTI Data Distribution Service* decide among the above two choices. If a shared memory transport plugin is available for local traffic, the effective value is 1 (i.e., disable UPV4 local traffic). Otherwise, the effective value is 0, i.e., use UDPv4 for local traffic also.
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.

Table 13.2 **Properties for the Builtin UDPv4 Transport**

Property Name (prefix with 'dds.transport.UDPv4.builtin.')	Property Value Description
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 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 *RTI Data Distribution Service* will use the zero-copy API if offered by the OS.
send_blocking	Controls the blocking behavior of send sockets. **CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS**. Currently two values are defined: NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS: Sockets are blocking (default socket options for Operating System). NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER: Sockets are modified to make them non-blocking. **This is not a supported configuration and may cause significant performance problems.**
transport_priority_mask	Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low and transport_priority_mapping_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.

Table 13.2 **Properties for the Builtin UDPv4 Transport**

<table>
<tr><th>Property Name
(prefix with
'dds.transport.UDPv4.builtin.')</th><th>Property Value Description</th></tr>
<tr><td>transport_priority_mapping_low</td><td rowspan="2">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>transport_priority_mapping_high</td></tr>
</table>

Table 13.3 **Properties for the Builtin UDPv6 Transport**

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
parent.address_bit_count	Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, this address_bit_count should be set to 8. For the range of addresses used by IPv4 (4 bytes), it should be set to 32.
parent.properties_bitmap	A bitmap that defines various properties of the transport to the *RTI Data Distribution Service* core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when *RTI Data Distribution Service* tries to receive a message using the plugin. This is in support of a zero-copy interface.

Table 13.3 **Properties for the Builtin UDPv6 Transport**

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
parent.gather_send_buffer_count_max	Specifies the maximum number of buffers that *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* from trying to gather too many buffers into a send call for the transport plugin. *RTI Data Distribution Service* requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is defined as NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.
parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plugin. This value must be set before the transport plugin is registered, so that *RTI Data Distribution Service* can properly use the plugin. If you set this higher than the default, then the *DomainParticipant's* ReceiverPoolQosPolicy's **buffer_size** should also be changed. See also: Note on page 13-9.
parent.allow_interfaces_list	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 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.

Table 13.3 **Properties for the Builtin UDPv6 Transport**

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
parent.deny_interfaces_list	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 "black" list is applied after the parent.allow_interfaces_list 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.
parent. allow_multicast_interfaces_list	A list of strings, each identifying a range of interface addresses. 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. This list sub-selects from the allowed interfaces that are obtained after applying the parent.allow_interfaces_list "white" list *and* the parent.deny_interfaces_list "black" list. Finally, the parent.deny_multicast_interfaces_list 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.

Table 13.3 **Properties for the Builtin UDPv6 Transport**

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
parent. deny_multicast_interfaces_list	A list of strings, each identifying a range of interface addresses. 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 "black" list is applied after the parent. allow_multicast_interfaces_list 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.
send_socket_buffer_size	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 set to NDDS_TRANSPORT_UDPV6_SOCKET_BUFFER_SIZE_OS_DEFAULT, then **setsockopt()** (or equivalent) will not be called to size the send buffer of the socket.
recv_socket_buffer_size	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 set to NDDS_TRANSPORT_UDPV6_SOCKET_-BUFFER_SIZE_OS_DEFAULT, then **setsockopt()** (or equivalent) will not be called to size the receive buffer of the socket.
unicast_enabled	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).

Table 13.3 **Properties for the Builtin UDPv6 Transport**

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
multicast_enabled	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).
multicast_ttl	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
multicast_loopback_disabled	Prevents the transport plugin from putting multicast packets onto the loopback interface. If disabled, then when sending multicast packets, *RTI Data Distribution Service* 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. **Note:** Windows CE does not support multicast loopback. This field is ignored for Windows CE targets.
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 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. Lets *RTI Data Distribution Service* decide among the above two choices. If a shared memory transport plugin is available for local traffic, the effective value is 1 (i.e., disable UDPv4 local traffic). Otherwise, the effective value is 0, i.e., use UDPv4 for local traffic also.

Table 13.3 **Properties for the Builtin UDPv6 Transport**

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
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.
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. By default this is set to 0, so *RTI Data Distribution Service* will use the zero-copy API if offered by the OS.
send_blocking	Controls the blocking behavior of send sockets. **CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.** Currently two values are defined: NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS: Sockets are blocking (default socket options for Operating System). NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER: Sockets are modified to make them non-blocking. **This is not a supported configuration and may cause significant performance problems.**

Table 13.3 **Properties for the Builtin UDPv6 Transport**

Property Name (prefix with 'dds.transport.UDPv6.builtin.')	Description
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 domain participant.
transport_priority_mask	Sets a mask for use of transport priority field. If transport priority mapping is supported on the platform[a], 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 TRANSPORT_PRIORITY QosPolicy (Section 6.5.19) 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 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 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 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	

a. Please refer to the Platform Notes to find out if the transport priority is supported on a specific platform.

Table 13.4 **Properties for the Builtin Shared Memory Transport**

Property Name (prefix with 'dds.transport.shmem.builtin.')	Property Value Description
parent.address_bit_count	Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, this address_bit_count should be set to 8. For the range of addresses used by IPv4 (4 bytes), it should be set to 32.

Table 13.4 **Properties for the Builtin Shared Memory Transport**

Property Name (prefix with 'dds.transport.shmem.builtin.')	**Property Value Description**
parent.properties_bitmap	A bitmap that defines various properties of the transport to the *RTI Data Distribution Service* core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* from trying to gather too many buffers into a send call for the transport plugin. *RTI Data Distribution Service* requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.
parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plugin. This value must be set before the transport plugin is registered, so that *RTI Data Distribution Service* can properly use the plugin. If you set this higher than the default, then the *DomainParticipant's* ReceiverPoolQosPolicy's **buffer_size** should also be changed. See also: Note on page 13-9.

Table 13.4 **Properties for the Builtin Shared Memory Transport**

Property Name (prefix with 'dds.transport.shmem.builtin.')	Property Value Description
parent.allow_interfaces_list	Not applicable to the Shared Memory Transport
parent.deny_interfaces_list	
parent. allow_multicast_interfaces_list	
parent. deny_multicast_interfaces_list	
received_message_count_max	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 **received_message_count_max** messages of the maximum size set in parent.message_size_max. The total number of bytes that can be buffered for a RecvResource is actually controlled by receive_buffer_size.

Table 13.4 **Properties for the Builtin Shared Memory Transport**

Property Name (prefix with 'dds.transport.shmem.builtin.')	Property Value Description
receive_buffer_size	The total number of bytes that can be buffered in the receive queue. This number controls how much memory is allocated by the plugin for the receive queue (on a per RecvResource basis). The actual number of bytes allocated is: size = **receive_buffer_size** + **message_size_max** + **received_message_count_max** * *fixedOverhead* where *fixedOverhead* is some small number of bytes used by the queue data structure. If **receive_buffer_size** < **message_size_max** * **received_message_count_max**, then the transport plugin will not be able to store **received_message_count_max** messages of size **message_size_max**. If **receive_buffer_size** > **message_size_max** * **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 example, if the maximum message size is 64K bytes, and you configure the plugin to buffer at least 10 messages, then 640K bytes of memory would be needed if all messages were 64K bytes. Should this be desired, then **receive_buffer_size** should be set to 640K bytes. However, if the average message size is only 10K bytes, then you could set the **receive_buffer_size** to 100K bytes. This allows you to optimize the memory usage of the plugin for the average case and yet allow the plugin to handle the extreme case. The queue will always be able to hold 1 message of **message_size_max** bytes, regardless of the value of **receive_buffer_size**.

13.6.1 Notes Regarding Loopback and Shared Memory

By default, *RTI Data Distribution Service* uses shared memory to communicate with other *DomainParticipants* on the same node, and disables local traffic over the UDPv4 or UPDv6 loopback interface. Thus, by default, an *RTI Data Distribution Service* application with shared memory enabled will not communicate with other applications on the same node that don't have shared memory enabled.

For example, suppose you have three *RTI Data Distribution Service* applications on the same node. Shared memory is enabled on Applications A and B, but disabled on Application C. In this scenario, A and B will communicate with each other, but they will not communicate with C.

You can change this behavior by setting the "**ignore_loopback_interface**" field of the UDPv4 transport properties to 0 on Applications A and B. This will force *DomainParticipants* with shared memory enabled to also communicate over UDPv4 or UDPv6 loopback (and thus find Application C without using shared memory). Alternatively, you can disable shared memory on A and B via the TransportBuiltinQosPolicy.

13.6.2 Setting the Maximum Gather-Send Buffer Count for UDPv4 and UDPv6

To minimize memory copies, *RTI Data Distribution Service* 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 *RTI Data Distribution Service*'s ability to concatenate multiple 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 samples that can be concatenated to five or six.

To match this limitation, *RTI Data Distribution Service* sets the UDPv4 and UDPv6 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 UDPv4 or UDPv6 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 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*, samples are sent asynchronously by a separate thread. Samples may not be sent immediately, but may be queued instead, depending on the settings of the associated FlowController. If multiple 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 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.

13.7 Installing Additional Builtin Transport Plugins with register_transport()

After you create an instance of a transport plugin (see Section 13.4) , you have to register it.

The builtin transports (UDPv4, UDPv6, and Shared Memory) are implicitly registered by default (if they are enabled via the TRANSPORT_BUILTIN QosPolicy (DDS Extension) (Section 8.5.8)). 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 the APIs other than Java or .NET. If you are using Java or .NET, use the Property QosPolicy to install additional transport plugins.)

```
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)
```

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 Transport Aliases (Section 13.7.2).

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 Transport Network Addresses (Section 13.7.3).

Note: You must ensure that the transport plugin instance is only used by one *DomainParticipant* at a time. See Section 13.7.1.

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.

13.7.1 Transport Lifecycles

If you create and register a transport plugin with a *DomainParticipant*, you are responsible for deleting it by calling its destructor. Builtin transport plugins are automatically managed by *RTI Data Distribution Service* 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—once 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.

13.7.2 Transport Aliases

In order to use a transport plugin instance in an *RTI Data Distribution Service* application, it must be registered with a *DomainParticipant* using the **register_transport()** operation (Section 13.7). **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*. Preconfigured builtin 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 DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2)), and for *DataWriters* and *DataReaders* (see TRANSPORT_SELECTION QosPolicy (DDS Extension) (Section 6.5.20)).
- ❏ Multicast addresses on which to receive discovery messages (see **multicast_receive_addresses** in DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2)), and the multicast addresses and ports on which to receive user data (DDS_DataReaderQos::multicast).
- ❏ Unicast ports used for user data (see TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21)) 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 Enabling Entities (Section 4.1.2)). 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" that others.

13.7.3 Transport Network Addresses

The address bits *not* used by the transport plugin for its internal addressing constitute its network address bits.

In order for *RTI Data Distribution Service* to properly route the messages, each unicast interface in the *RTI Data Distribution Service* domain must have a unique address.

You specify the network address when installing a transport plugin via the **register_transport()** operation (Section 13.7). Choose the network address for a transport plugin so that the resulting fully qualified 128-bit address will be unique in the *RTI Data Distribution Service* 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).

13.8 Installing Additional Builtin Transport Plugins with PropertyQosPolicy

Similar to default builtin transport instances, additional builtin transport instances can also be configured through PROPERTY QosPolicy (DDS Extension) (Section 6.5.15).

To install additional instances of builtin transport, the Properties listed in Table 13.5 are required.

Table 13.5 **Properties for Dynamically Loading and Registering Additional Builtin Transport Plugins**

Property Name	Description
dds.transport.load_plugins	Comma-separated list of <TRANSPORT_PREFIX>. Up to <TRANSPORT_PREFIX> entries may be specified.
<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 Installing Additional Builtin Transport Plugins with register_transport() (Section 13.7)). Aliases should be specified as a comma separated string, with each comma delimiting an alias. If it is not specified, <TRANSPORT_PREFIX> is used as the default alias for the plugin.
<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 Installing Additional Builtin Transport Plugins with register_transport() (Section 13.7)). 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 13.2 on page 13-8 through Table 13.4 on page 13-21 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.<builitn_transport_name>.<instance_name>**, where **<instance_name>** is configured through the **dds.transport.load_plugins** property instead of **dds.transport.<builtin_transport_name>.builtin**.

13.9 Other Transport Support Operations

13.9.1 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.

```
DDS_ReturnCode_t NDDSTransportSupport::add_send_route
        (const NDDS_Transport_Handle_t &  transport_handle_in,
         const NDDS_Transport_Address_t &  address_range_in,
         DDS_Long  address_range_bit_count_in)
```

transport_handle_in A valid non-NIL transport handle as a result of a call to **register_transport()** (Section 13.7).

address_range_in The outgoing 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. You can set up a routing table to restrict the use of a transport plugin to send messages to selected addresses ranges.

```
+--------------------------------------------------------------------+
|         Outgoing Address Range 1     ->      Transport Plugin      |
+--------------------------------------------------------------------+
|              :                       ->             :              |
+--------------------------------------------------------------------+
|         Outgoing Address Range K     ->      Transport Plugin      |
+--------------------------------------------------------------------+
```

13.9.2 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.

```
DDS_ReturnCode_t NDDSTransportSupport::add_receive_route
      (const NDDS_Transport_Handle_t &  transport_handle_in,
       const NDDS_Transport_Address_t &  address_range_in,
       DDS_Long  address_range_bit_count_in)
```

transport_handle_in A valid non-NIL transport handle as a result of a call to **register_transport()** (Section 13.7).

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  |
+-----------------------------------------------------------------+
|                :                    <-          :                 |
+-----------------------------------------------------------------+
|          Transport Plugin           <-  Incoming Address Range M  |
+-----------------------------------------------------------------+
```

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).

13.9.3 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.

```
NDDS_Transport_Handle_t NDDSTransportSupport::lookup_transport
            (DDSDomainParticipant *  participant_in,
             DDS_StringSeq &  aliases_out,
             NDDS_Transport_Address_t &  network_address_out,
             NDDS_Transport_Plugin *  transport_in )
```

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.

13. Transport Plugins

Chapter 14 Built-In Topics

This chapter discusses how to use Built-in Topics.

RTI Data Distribution Service must discover and keep track of remote entities, such as new participants in the domain. This information may also be important to the application itself, which may want to react to this discovery, or else access it on demand. To support these needs, *RTI Data Distribution Service* provides *built-in Topic*s ("DCPSParticipant", "DCPSPublication", "DCPSSubscription" in Figure 12.2 on page 12-11) 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.

14.1 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 builtin *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,” on page 4-29 and “Hierarchical Processing of Listeners” on page 4-30.

14.2 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 *DataReader*s 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.

Table 14.1 through Table 14.4 describe the built-in topics and their data types. The USER_DATA QosPolicy (Section 6.5.23), TOPIC_DATA QosPolicy (Section 5.2.1) and GROUP_DATA QosPolicy (Section 6.4.4) are included as part of the built-in data type and are not used by *RTI Data Distribution Service*. 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 Section 14.4).

Table 14.1 **Participant Built-in Topic's Data Type (DDS_ParticipantBuiltinTopicData)**

Type	Field	Description
DDS_BuiltinTopicKey	key	DDS key to distinguish the discovered *DomainParticipant*
DDS_UserDataQosPolicy	user_data	Data that can be set when the related *DomainParticipant* is created (via the USER_DATA QosPolicy (Section 6.5.23)) and that the application may use as it wishes (e.g., to perform some security checking).
DDS_PropertyQosPolicy	property	Pairs of names/values to be stored with the *DomainParticipant*. See PROPERTY QosPolicy (DDS Extension) (Section 6.5.15). The usage is strictly application-dependent.
DDS_ProtocolVersion_t	rtps_protocol_version	Version number of the RTPS wire protocol used.
DDS_VendorId_t	rtps_vendor_id	ID of vendor implementing the RTPS wire protocol.
DDS_UnsignedLong	dds_builtin_endpoints	A bitmap set by the discovery plugins. Each bit in this field indicates a builtin endpoint present for discovery.
DDS_LocatorSeq	default_unicast_locators	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/Subscription-BuiltinTopicData is empty, the default_unicast_locators in the corresponding Participant Builtin Topic Data is assumed. If default_unicast_locators is empty, it defaults to DomainParticipantQos.default_unicast.
DDS_ProductVersion_t	product_version	A vendor-specific parameter. The current version of *RTI Data Distribution Service*.
DDS_EntityNameQosPolicy	participant_name	A name assigned to the *DomainParticipant*. See ENTITYNAME QosPolicy (DDS Extension) (Section 8.5.5).

Table 14.2 **Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)**

Type	Field	Description
DDS_BuiltinTopicKey_t	key	DDS key to distinguish the discovered *DataWriter*
DDS_BuiltinTopicKey_t	participant_key	DDS key to distinguish the participant to which the discovered *DataWriter* belongs

Table 14.2 **Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)**

Type	Field	Description
DDS_String	topic_name	Topic name of the discovered *DataWriter*
DDS_String	type_name	Type name attached to the topic of the discovered *DataWriter*
DDS_DurabilityQosPolicy	durability	QosPolicies of the discovered *DataWriter*
DDS_DurabilityService-QosPolicy	durability_service	
DDS_DeadlineQosPolicy	deadline	
DDS_DestinationOrder-QosPolicy	destination_order	
DDS_LatencyBudget-QosPolicy	latency_budget	
DDS_LivelinessQosPolicy	liveliness	
DDS_ReliabilityQosPolicy	reliability	
DDS_LifespanQosPolicy	lifespan	
DDS_UserDataQosPolicy	user_data	Data that can be set when the *DataWriter* is created (via the USER_DATA QosPolicy (Section 6.5.23)) and that the application may use as it wishes.
DDS_OwnershipQosPolicy	ownership	QosPolicies of the discovered *DataWriter*
DDS_OwnershipStrength-QosPolicy	ownership_strength	
DDS_DestinationOrder-QosPolicy	destination_order	
DDS_PresentationQosPolicy	presentation	
DDS_PartitionQosPolicy	partition	Name of the partition, set in the PARTITION QosPolicy (Section 6.4.5) for the publisher to which the discovered *DataWriter* belongs
DDS_TopicDataQosPolicy	topic_data	Data that can be set when the *Topic* (with which the discovered *DataWriter* is associated) is created (via the TOPIC_DATA QosPolicy (Section 5.2.1)) and that the application may use as it wishes.
DDS_GroupDataQosPolicy	group_data	Data that can be set when the *Publisher* to which the discovered *DataWriter* belongs is created (via the GROUP_DATA QosPolicy (Section 6.4.4)) and that the application may use as it wishes.

Table 14.2 **Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)**

Type	Field	Description
DDS_TypeCode *	type_code	Type code information about this *Topic*. See Using Generated Types without RTI Data Distribution Service (Standalone) (Section 3.7).
DDS_BuiltinTopicKey_t	publisher_key	The key of the *Publisher* to which the *DataWriter* belongs.
DDS_PropertyQosPolicy	property	Properties (pairs of names/values) assigned to the corresponding *DataWriter*. Usage is strictly application-dependent. See PROPERTY QosPolicy (DDS Extension) (Section 6.5.15).
DDS_LocatorSeq	unicast_locators	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.
DDS_GUID_t	virtual_guid	Virtual GUID for the corresponding *DataWriter*. For more information, see Durability and Persistence Based on Virtual GUIDs (Section 11.2).
DDS_ProtocolVersion_t	rtps_protocol_ version	Version number of the RTPS wire protocol in use.
DDS_VendorId_t	rtps_vendor_id	ID of the vendor implementing the RTPS wire protocol.
DDS_Product_Version_t	product_version	A vendor-specific value. For RTI, this is the current version of *RTI Data Distribution Service*.

Table 14.2 **Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)**

Type	Field	Description
DDS_Boolean	disable_positive_acks	This is a vendor specific parameter. Determines whether matching *DataReaders* send positive acknowledgements for reliability.
DDS_LocatorFilterQosPolicy	locator_filter	When the MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12) 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 LOCATOR_FILTER QoS Policy (DDS Extension) (Section 14.2.1).

Table 14.3 **Subscription Built-in Topic's Data Type (DDS_SubscriptionBuiltinTopicData)**

Type	Field	Description
DDS_BuiltinTopicKey_t	key	DDS key to distinguish the discovered *DataReader.*
DDS_BuiltinTopicKey_t	participant_key	DDS key to distinguish the participant to which the discovered *DataReader* belongs.
char *	topic_name	Topic name of the discovered *DataReader.*
char *	type_name	Type name attached to the *Topic* of the discovered *DataReader.*
DDS_DurabilityQosPolicy	durability	QosPolicies of the discovered *DataReader*
DDS_DeadlineQosPolicy	deadline	
DDS_LatencyBudgetQosPolicy	latency_budget	
DDS_LivelinessQosPolicy	liveliness	
DDS_ReliabilityQosPolicy	reliability	
DDS_OwnershipQosPolicy	ownership	
DDS_DestinationOrderQosPolicy	destination_order	
DDS_UserDataQosPolicy	user_data	Data that can be set when the *DataReader* is created (via the USER_DATA QosPolicy (Section 6.5.23)) and that the application may use as it wishes.

Table 14.3 **Subscription Built-in Topic's Data Type (DDS_SubscriptionBuiltinTopicData)**

Type	Field	Description
DDS_ TimeBasedFilterQosPolicy	time_based_filter	QosPolicies of the discovered *DataReader*
DDS_PresentationQosPolicy	presentation	
DDS_PartitionQosPolicy	partition	Name of the partition, set in the PARTITION QosPolicy (Section 6.4.5) for the *Subscriber* to which the discovered *DataReader* belongs.
DDS_TopicDataQosPolicy	topic_data	Data that can be set when the *Topic* to which the discovered *DataReader* belongs is created (via the TOPIC_DATA QosPolicy (Section 5.2.1)) and that the application may use as it wishes.
DDS_GroupDataQosPolicy	group_data	Data that can be set when the *Publisher* to which the discovered *DataReader* belongs is created (via the GROUP_DATA QosPolicy (Section 6.4.4)) and that the application may use as it wishes.
DDS_TypeCode *	type_code	Type code information about this *Topic*. See Using Generated Types without RTI Data Distribution Service (Standalone) (Section 3.7).
DDS_BuiltinTopicKey_t	subscriber_key	Key of the *Subscriber* to which the *DataReader* belongs.
DDS_PropertyQosPolicy	property	Properties (pairs of names/values) assigned to the corresponding *DataReader*. Usage is strictly application-dependent. See PROPERTY QosPolicy (DDS Extension) (Section 6.5.15).
DDS_LocatorSeq	unicast_locators	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.
DDS_LocatorSeq	multicast_locators	Custom multicast locators that the endpoint can specify.
DDS_ContentFilter-Property_t	content_filter_property	Provides all the required information to enable content filtering on the writer side.

Table 14.3 **Subscription Built-in Topic's Data Type (DDS_SubscriptionBuiltinTopicData)**

Type	Field	Description
DDS_GUID_t	virtual_guid	Virtual GUID for the corresponding *DataReader*. For more information, see Durability and Persistence Based on Virtual GUIDs (Section 11.2).
DDS_ProtocolVersion_t	rtps_protocol_version	Version number of the RTPS wire protocol in use.
DDS_VendorId_t	rtps_vendor_id	ID of the vendor implementing the RTPS wire protocol.
DDS_Product_Version_t	product_version	A vendor-specific value. For RTI, this is the current version of *RTI Data Distribution Service*.
DDS_Boolean	disable_positive_acks	This is a vendor specific parameter. Determines whether matching *DataReaders* send positive acknowledgements for reliability.

Table 14.4 **Topic Built-in Topic's Data Type (DDS_TopicBuiltinTopicData) (**See "Note:" on page 14-11**)**

Type	Field	Description
DDS_BuiltinTopicKey_t	key	DDS key to distinguish the discovered *Topic*
DDS_String	name	*Topic* name
DDS_String	type_name	type name attached to the *Topic*
DDS_DurabilityQosPolicy	durability	QosPolicy of the discovered *Topic*
DDS_DurabilityServiceQosPolicy	durability_service	
DDS_DeadlineQosPolicy	deadline	
DDS_LatencyBudgetQosPolicy	latency_budget	
DDS_LivelinessQosPolicy	liveliness	
DDS_ReliabilityQosPolicy	reliability	
DDS_TransportPriorityQosPolicy	transport_priority	
DDS_LifespanQosPolicy	lifespan	
DDS_DestinationOrderQosPolicy	destination_ order	
DDS_HistoryQosPolicy	history	
DDS_ResourceLimitsQosPolicy	resource_limits	
DDS_OwnershipQosPolicy	ownership	
DDS_TopicDataQosPolicy	topic_data	Data that can be set when the *Topic* to which the discovered *DataReader* belongs is created (via the TOPIC_DATA QosPolicy (Section 5.2.1)) and that the application may use as it wishes.

Table 14.5 lists the QoS of the built-in Subscriber and DataReader that are created for accessing discovery data. These are provided for your reference only; they cannot be changed.

Table 14.5 **QoS of Built-in Subscriber and DataReader**

QosPolicy	Value
Deadline	period = infinite
DestinationOrder	kind = BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS
Durability	kind = TRANSIENT_LOCAL_DURABILITY_QOS
EntityFactory	autoenable_created_entities = TRUE
GroupData	value = empty sequence
History	kind = KEEP_LAST_HISTORY_QOS depth = 1
LatencyBudget	duration = 0
Liveliness	kind = AUTOMATIC_LIVELINESS_QOS lease_duration = infinite
Ownership	kind = SHARED_OWNERSHIP_QOS
Ownership Strength	value = 0
Presentation	access_scope = TOPIC_PRESENTATION_QOS coherent_access = FALSE ordered_access = FALSE
Partition	name = empty sequence
ReaderDataLifecycle	autopurge_nowriter_samples_delay = infinite
Reliability	kind = RELIABLE_RELIABILITY_QOS max_blocking_time is irrelevant for the *DataReader*
ResourceLimits	Depends on setting of DomainParticipantResourceLimitsQosPolicy and DiscoveryConfigQosPolicy in DomainParticipantQos: max_samples = domainParticipantQos.discovery_config.[participant/publication/subscription]_reader_resource_limits.max_samples max_instances = domainParticipantQos.resource_limits.[remote_writer/reader/participant]_allocation.max_count max_samples_per_instance = 1
TimeBasedFilter	minimum_separation = 0
TopicData	value = empty sequence
UserData	value = empty sequence

Note: The DDS_TopicBuiltinTopicData built-in topic (described in Table 14.4) 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 builtin *DataReaders*.

14.2.1 LOCATOR_FILTER QoS Policy (DDS Extension)

The LocatorFilter QoS Policy is only applicable to the builtin topic for a Publication (see Table 14.2, "Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)," on page 14-3).

Table 14.6 **DDS_LocatorFilterQosPolicy**

Type	Field Name	Description
DDS_LocatorFilterSeq	locator_filters	A sequence of locator filters, described in Table 14.7 on page 14-11. There is one locator filter per *DataWriter* channel. If the length of the sequence is zero, the *DataWriter* is not using multi-channel.
char *	filter_name	Name of the filter class used to describe the locator filter expressions. The following two values are supported: • DDS_SQLFILTER_NAME • DDS_STRINGMATCHFILTER_NAME

Table 14.7 **DDS_LocatorFilter_t**

Type	Field Name	Description
DDS_LocatorSeq	locators	A sequence of multicast address locators for the locator filter. See Table 14.8 on page 14-12.
char *	filter_expression	A logical expression used to determine if the data will be published in the channel associated with this locator filter. See "SQL Filter Expression Notation" on page 5-23 and "STRINGMATCH Filter Expression Notation" on page 5-32 for information about the expression syntax.

Table 14.8 **DDS_Locator_t**

Type	Field Name	Description
DDS_Long	kind	If the locator kind is DDS_LOCATOR_KIND_UDPv4[a], 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[a], 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.
DDS_Octet[16]	address	The locator address.
DDS_UnsignedLong	port	The locator port number.

a. In C#, the locator kinds for UDPv4 and UDPv6 addresses are Locator_t.LOCATOR_KIND_UDPv4 and Locator_t.LOCATOR_KIND_UDPv6.

14.3 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.

```
// Lookup builtin reader
DDSDataReader *builtin_reader =
builtin_subscriber->lookup_datareader(DDS_PUBLICATION_TOPIC_NAME);

if (builtin_reader == NULL) {
    // ... error
}

// Register listener to builtin reader
MyPublicationBuiltinTopicDataListener builtin_reader_listener =
                  new MyPublicationBuiltinTopicDataListener();

if (builtin_reader->set_listener(builtin_reader_listener,
                  DDS_DATA_AVAILABLE_STATUS) != DDS_RETCODE_OK) {
    // ... error
}

// enable domain participant
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.

14.4 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.

```
DDS_ReturnCode_t  ignore_participant (const DDS_InstanceHandle_t &handle)
DDS_ReturnCode_t  ignore_publication (const DDS_InstanceHandle_t &handle)
DDS_ReturnCode_t  ignore_subscription (const DDS_InstanceHandle_t &handle)
DDS_ReturnCode_t  ignore_topic (const DDS_InstanceHandle_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 USER_DATA QosPolicy (Section 6.5.23). 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 USER_DATA QosPolicy (Section 6.5.23), GROUP_DATA QosPolicy (Section 6.4.4) or TOPIC_DATA QosPolicy (Section 5.2.1). 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: Chapter 12: Discovery.

14.4.1 Ignoring Specific Remote DomainParticipants

The **ignore_participant()** operation is used to instruct *RTI Data Distribution Service* to locally ignore a remote *DomainParticipant*. It causes *RTI Data Distribution Service* to locally behave as if the remote *DomainParticipant* does not exist.

```
DDS_ReturnCode_t ignore_participant  (const DDS_InstanceHandle_t &  handle)
```

After invoking this operation, *RTI Data Distribution Service* will locally ignore any *Topic*, *publication*, or *subscription* that originates on that *DomainParticipant*. (If you only want to ignore specific publications or subscriptions, see Section 14.4.2 instead.) Figure 14.1, "Ignoring Participants," on page 14-15 provides an example.

Caution: There is no way to reverse this operation. You can add to the peer list, however—see Section 8.5.2.3.

14.4.2 Ignoring Publications and Subscriptions

You can instruct *RTI Data Distribution Service* 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:

Figure 14.1 **Ignoring Participants**

```
class MyParticipantBuiltinTopicDataListener : public DDSDataReaderListener {
    public:
        virtual void on_data_available(DDSDataReader *reader);
        // ......
};
void MyParticipantBuiltinTopicdataListener::on_data_available(
DDSDataReader *reader) {

       DDSParticipantBuiltinTopicDataReader *builtinTopicDataReader =
             DDSParticipantBuiltinTopicDataDataReader *) reader;

       DDS_ParticipantBuiltinTopicDataSeq data_seq;
       DDS_SampleInfoSeq info_seq;
       int = 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
       }
}
```

- ❑ 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 data samples available for the entity's built-in *DataReader*.

```
DDS_ReturnCode_t ignore_publication  (const DDS_InstanceHandle_t &  handle)
DDS_ReturnCode_t ignore_subscription (const DDS_InstanceHandle_t &  handle)
```

Caution: There is no way to reverse these operations.

Figure 14.2, "Ignoring Publications," on page 14-17 provides an example.

14.4.3 Ignoring Topics

The **ignore_topic()** operation instructs *RTI Data Distribution Service* to locally ignore a *Topic*. This means it will locally ignore any publication or subscription to the *Topic*.

```
DDS_ReturnCode_t ignore_topic (const DDS_InstanceHandle_t &  handle)
```

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 data samples from the built-in DataReader to the *Topic*.

Figure 14.2 **Ignoring Publications**

```
class MyPublicationBuiltinTopicDataListener : public DDSDataReaderListener {
    public:
        virtual void on_data_available(DDSDataReader *reader);
        // ......
};

void MyPublicationBuiltinTopicdataListener::on_data_available(
                                   DDSDataReader *reader) {

       DDSPublicationBuiltinTopicDataReader *builtinTopicDataReader =
                             (DDS_PublicationBuiltinTopicDataReader *)reader;
       DDS_PublicationBuiltinTopicDataSeq data_seq;
       DDS_SampleInfoSeq info_seq;
       int = 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) {
             // ... error
       }
}
```

14. Built-In Topics

Chapter 15 Configuring QoS with XML

RTI Data Distribution Service entities are configured by means of QosPolicies. The QoS 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.
- ❏ 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 *RTI Data Distribution Service* entities using XML:

- ❏ Example XML File (Section 15.1)
- ❏ How to Load XML-Specified QoS Settings (Section 15.2)
- ❏ How to Use XML-Specified QoS Settings (Section 15.3)
- ❏ XML File Syntax (Section 15.4)
- ❏ XML String Syntax (Section 15.5)
- ❏ How the XML is Validated (Section 15.6)
- ❏ Configuring QoS with XML (Section 15.7)
- ❏ QoS Profiles (Section 15.8)
- ❏ QoS Libraries (Section 15.9)
- ❏ URL Groups (Section 15.10)

15.1 Example XML File

The QoS configuration of a DDS entity can be loaded from an XML file or string. 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:

Example Configuration File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A XML configuration file -->
<dds version = 4.5x>
  <qos_library name="RTILibrary">
      <!--Individual QoS are a shortcut for QoS Profiles with single QoS->
      <datawriter_qos name="KeepAllWriter">
            <history>
                <kind>KEEP_ALL_HISTORY_QOS</kind>
            </history>
      </datawriter_qos>
      <!-- A Qos Profile is a set of related QoS -->
      <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>
  </qos_library>
</dds>
```

See **$NDDSHOME/resource/qos_profiles_4.5x**[1]**/xml/NDDS_QOS_PROFILES.example.xml** for another example; this file contains the default QoS values for all entity kinds.

1. *x* stands for the version letter of the current release.

15.2 How to Load XML-Specified QoS Settings

If specified, XML-specified QoS settings are automatically loaded by the DomainParticipantFactory. There are several ways to load XML QoS profiles into your application.

The following list presents the various approaches, listed by load order:

- **$NDDSHOME/resource/qos_profiles_4.5x[1]/xml/NDDS_QOS_PROFILES.xml**
  This file is loaded automatically *if it exists* (not the default) *and* **ignore_resource_profile** in the PROFILE QosPolicy (DDS Extension) (Section 8.4.1) is FALSE (the default). **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. (*First to be loaded*)
- **URL Groups in NDDS_QOS_PROFILES**
  URL groups (see URL Groups (Section 15.10)) separated by semicolons referenced by the environment variable **NDDS_QOS_PROFILES** are loaded automatically if they exist *and* **ignore_environment_profile** in PROFILE QosPolicy (DDS Extension) (Section 8.4.1) is FALSE (the default).
- **<working directory>/USER_QOS_PROFILES.xml**
  This file is loaded automatically if it exists *and* **ignore_user_profile** in PROFILE QosPolicy (DDS Extension) (Section 8.4.1) is FALSE (the default).
- **URL groups in url_profile**
  URL groups (see URL Groups (Section 15.10)) referenced by **url_profile** (in PROFILE QosPolicy (DDS Extension) (Section 8.4.1)) will be loaded automatically if specified.
- **XML strings in string_profile**
  The sequence of XML strings referenced by **string_profile** (in PROFILE QosPolicy (DDS Extension) (Section 8.4.1)) will be loaded automatically if specified. (*Last to be loaded*)

You may use a combination of the above approaches.

1. *x* stands for the version letter of the current release.

The location of the XML documents (only files and strings are supported) is specified using URL (Uniform Resource Locator) format. 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, *RTI Data Distribution Service* will assume a file name. For example:

- ❑ File Specification: **/usr/local/default_dds.xml**

Duplicate QoS profiles are not allowed. *RTI Data Distribution Service* will report an error message in these scenarios. To overwrite a QoS profile, use QoS-Profile Inheritance (Section 15.8.2).

15.2.1 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 Section 8.3.1)
- ❑ **create_participant_with_profile()** (see Section 8.3.1)
- ❑ **get_*<entity>*_qos_from_profile()** (where *<entity>* is participant, topic, publisher, subscriber, datawriter, or datareader) (see Section 8.2.5)
- ❑ **get_*<entity>*_qos_from_profile_w_topic_name()** (where *<entity>* is topic, datawriter, or datareader) (see Section 8.2.5)
- ❑ **get_default_participant_qos()** (see Section 8.2.2)
- ❑ **get_qos_profile_libraries()** (See Section 15.9.1)
- ❑ **get_qos_profiles()** (See Section 15.8.5)
- ❑ **load_profiles()**
- ❑ **set_default_participant_qos_with_profile()** (see Section 8.2.2)
- ❑ **set_default_library()** (see Section 6.2.4.3)
- ❑ **set_default_profile()** (see Section 6.2.4.3)

QoS profiles are *reloaded* when either of these DomainParticipantFactory operations are called:

- ❑ **reload_profiles()**
- ❑ **set_qos()** (see Section 4.1.7)

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.

```
DDS_ReturnCode_t unload_profiles()
```

15.3 How to Use XML-Specified QoS Settings

You can use the operations listed in Table 15.1 to refer and use QoS profiles (see Section 15.8) described in XML files and XML strings.

Table 15.1 **Operations for Working with QoS Profiles**

Working With ...	Profile-Related Operations	Reference
DataReaders	set_qos_with_profile	Section 7.3.7.2
DataWriters	set_qos_with_profile	Section 6.3.12.2

Table 15.1 **Operations for Working with QoS Profiles**

Working With ...	Profile-Related Operations	Reference
DomainParticipants	create_datareader_with_profile	Section 7.3.1
	create_datawriter_with_profile	Section 6.3.1
	create_publisher_with_profile	Section 6.2.2
	create_subscriber_with_profile	Section 7.2.2
	create_topic_with_profile	Section 5.1.1
	get_default_library	Section 8.3.6.3
	get_default_profile	
	get_default_profile_library	
	set_default_datareader_qos_with_profile	Section 8.3.6.4
	set_default_datawriter_qos_with_profile	
	set_default_library	Section 8.3.6.3
	set_default_profile	
	set_default_publisher_qos_with_profile	Section 8.3.6.4
	set_default_subscriber_qos_with_profile	
	set_default_topic_qos_with_profile	
	set_qos_with_profile	Section 8.3.6.2

Table 15.1 **Operations for Working with QoS Profiles**

Working With ...	Profile-Related Operations	Reference
DomainParticipantFactory	create_participant_with_profile	Section 8.3.1
	get_datareader_qos_from_profile	Section 8.2.5
	get_datawriter_qos_from_profile	
	get_datawriter_qos_from_profile_w_topic_name	
	get_datareader_qos_from_profile_w_topic_name	
	get_default_library	Section 8.2.1.1
	get_default_profile	
	get_default_profile_library	
	get_participant_qos_from_profile	Section 8.2.5
	get_publisher_qos_from_profile	
	get_subscriber_qos_from_profile	
	get_topic_qos_from_profile	
	get_topic_qos_from_profile_w_topic_name	
	get_qos_profiles	Section 15.8.5
	get_qos_profile_libraries	Section 15.9.1
	load_profiles	Section 15.2.1
	reload_profiles	
	set_default_participant_qos_with_profile	Section 8.2.2
	set_default_library	Section 8.2.1.1
	set_default_profile	
	unload_profiles	Section 15.2.1
Publishers	create_datawriter_with_profile	Section 6.2.2
	get_default_library	Section 6.2.4.3
	get_default_profile	
	get_default_profile_library	
	set_default_datawriter_qos_with_profile	Section 6.2.4.4
	set_default_library	Section 6.2.4.3
	set_default_profile	
	set_qos_with_profile	Section 6.2.4.2

Table 15.1 **Operations for Working with QoS Profiles**

Working With ...	Profile-Related Operations	Reference
Subscribers	create_datareader_with_profile	Section 7.3.1
	get_default_library	Section 7.2.4.3
	get_default_profile	
	get_default_profile_library	
	set_default_datawriter_qos_with_profile	Section 7.2.4.4
	set_default_library	Section 7.2.4.3
	set_default_profile	
	set_qos_with_profile	Section 7.2.4.2
Topics	set_qos_with_profile	Section 5.1.3

15.4 XML File Syntax

The XML 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 tag value is a UTF-8 encoded string. Legal values are alphanumeric characters. The middleware's parser will remove all leading and trailing spaces[1] 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 15.2.

1. 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 (see Section 15.6).

Table 15.2 **Supported Tag Values**

Type	Format	Notes
DDS_Boolean	yes[a], 1, true, BOOLEAN_TRUE[a] or DDS_BOOLEAN_TRUE[a]: these all mean TRUE	Not case-sensitive
	no[a], 0, false, BOOLEAN_FALSE[a] or DDS_BOOLEAN_FALSE[a]: these all mean FALSE	
DDS_Enum	A string. Legal values are those listed in the online (HTML) documentation for the C[a] or Java API.	Must be specified as a string. (Do not use numeric values.)
DDS_Long	-2147483648 to 2147483647 or 0x80000000 to 0x7fffffff[a] or LENGTH_UNLIMITED or DDS_LENGTH_UNLIMITED[a]	A 32-bit signed integer
DDS_UnsignedLong	0 to 4294967296 or 0 to 0xffffffff[a]	A 32-bit unsigned integer
String	UTF-8 character string	All leading and trailing spaces are ignored between two tags

a. These values will not be considered valid if you use the distributed XSD document to do validation at run-time with a code editor (see Section 15.6).

15.5 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 DDS 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 PROFILE QosPolicy (DDS Extension) (Section 8.4.1). Each string URL must contain a whole XML document.

- ❏ The **string_profile** field in the PROFILE QosPolicy (DDS Extension) (Section 8.4.1) 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.

15.6 How the XML is Validated

15.6.1 Validation at Run-Time

RTI Data Distribution Service validates the input XML files using a builtin Document Type Definition (DTD).

You can find a copy of the builtin DTD in **$(NDDSHOME)/resource/qos_profiles_4.5x**[1]**/schema/rti_dds_qos_profiles.dtd**. (This is only a *copy* of what the *RTI Data Distribution Service* 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 *RTI Data Distribution Service* must use a DTD file from a user's directory to perform validation:

```
<!DOCTYPE dds SYSTEM "/local/joe/rti/dds/mydds.dtd">
```

- ❏ 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 builtin DTD is used.

1. *x* stands for the version letter of the current release.

❑ The XML files used by *RTI Data Distribution Service* can be versioned using the attribute version in the <dds> tag. For example:

```
<dds version="4.5x">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 *RTI Data Distribution Service* 4.5x tries to load an XML file from *RTI Data Distribution Service* 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 RTI Data Distribution Service (4.5x). If
these versions are not compatible, that incompatibility could be
the cause of this error
```

15.6.2 XML File Validation During Editing

RTI Data Distribution 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 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 **$(NDDSHOME)/resource/qos_profiles_4.5x/schema/rti_dds_qos_profiles.dtd** and **$(NDDSHOME)/resource/qos_profiles_4.5x/schema/rti_dds_qos_profiles.xsd**, respectively (in '4.5*x*', *x* stands for the version letter of the current release).

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=
"<NDDSHOME2>/resource/qos_profiles_4.5x/schema/rti_dds_qos_profiles.xsd">
    ...
</dds>
```

1. *x* stands for the version letter of the current release.
2. Replace <NDDSHOME> with the installation directory of *RTI Data Distribution Service*.

15. Configuring QoS with XML

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dds SYSTEM
"<NDDSHOME2>/resource/qos_profiles_4.5x/schema/rti_dds_qos_profiles.dtd">
<dds>
    ...
</dds>
```

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.

15.7 Configuring QoS with XML

To configure the QoS for a DDS Entity using XML, use the following tags:

- <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:

```
<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 Topic Filters (Section 15.8.3) and QoS Profiles (Section 15.8).

15.7.1 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:

```
struct DDS_Duration_t {
    DDS_Long sec;
    DDS_UnsignedLong nanosec;
}
struct DDS_ReliabilityQosPolicy {
    DDS_ReliabilityQosPolicyKind kind;
    DDS_Duration_t max_blocking_time;
}
```

The equivalent representation in XML is as follows:

```
<reliability>
    <kind></kind>
    <max_blocking_time>
           <sec></sec>
           <nanosec></nanosec>
    </max_blocking_time>
</reliability>
```

15.7.2 Sequences

In general, sequences in QosPolicies are described with the following XML format:

```
<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:

```
<property>
    <value>
        <element>
            <name>my name</name>
            <value>my value</value>
        </element>
        <element>
            <name>my name2</name>
            <value>my value2</value>
        </element>
    </value>
</property>
```

A sequence without elements represents a sequence of length 0. For example:

```
<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) (Section 8.5.2)), 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:

```
<discovery>
    <!-- initial_peers sequence contains the default value -->
    <!-- <initial_peers/> -->
</discovery>
```

Sequences defined in a derived[1] QoS will replace the corresponding sequences in the base QoS. For example:

```
  <participant_qos name="MyBaseQos">
    <property>
        <value>
            <element>
                <name>
                 dds.transport.UDPv4.builtin.send_socket_buffer_size
                </name>
                <value>524288</value>
            </element>
```

1. The concepts of *derived* and *base* QoS are described in QoS-Profile Inheritance (Section 15.8.2).

```
                <element>
                    <name>
                     dds.transport.UDPv4.builtin.recv_socket_buffer_size
                    </name>
                    <value>2097152</value>
                </element>
            </value>
        </property>
    </participant_qos>

    <participant_qos name="MyDerivedQos" base_name="MyBaseQos">
        <property>
            <value>
                <element>
                    <name>
                     dds.transport.UDPv4.builtin.recv_socket_buffer_size
                    </name>
                    <value>1048576</value>
                </element>
            </value>
        </property>
    </qos_profile>
```

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.

As a special case, sequences of octets are represented with a single XML tag enclosing a sequence of decimal/hexadecimal values between 0..255 separated with commas. For example:

```
<user_data>
    <value>100,200,0,0,0,223</value>
</user_data>

<topic_data>
    <value>0xff,0x00,0x8e,0xEE,0x78</value>
</topic_data>
```

15.7.3 Arrays

In general, the arrays contained in the QosPolicies are described with the following XML format:

```
<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:

```
<reader_qos>
    ...
    <protocol>
        <virtual_guid>
            <value>1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16</value>
        </virtual_guid>
    </protocol>
</reader_qos>
```

15.7.4 Enumeration Values

Enumeration values are represented using their C or Java string representation. For example:

```
<history>
    <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
</history>
```

or

```
<history>
    <kind>KEEP_ALL_HISTORY_QOS</kind>
</history>
```

When the XSD document is used for validation during editing (see Section 15.6.2), only the Java representation is valid.

15.7.5 Time Values (Durations)

You can use the following special values for fields that require seconds or nanoseconds:

- ❑ DURATION_INFINITE_SEC or DDS_DURATION_INFINITE_SEC,
- ❑ DURATION_ZERO_SEC or DDS_DURATION_ZERO_SEC,
- ❑ DURATION_INFINITE_NSEC or DDS_DURATION_INFINITE_NSEC
- ❑ DURATION_ZERO_NSEC or DDS_DURATION_ZERO_NSEC

For example:

```
<deadline>
    <period>
        <sec>DURATION_INFINITE_SEC</sec>
        <nanosec>DURATION_INFINITE_NSEC</nanosec>
    </period>
</deadline>
```

When the XSD document is used for validation during editing (see Section 15.6.2), only the values without the DDS prefix are considered valid.

15.7.6 Transport Properties

You can configure transport plugins using the *DomainParticipant's* PROPERTY QosPolicy (DDS Extension) (Section 6.5.15).

- ❑ Properties for the builtin transports are described in Setting Builtin Transport Properties with the PropertyQosPolicy (Section 13.6).
- ❑ Properties for other transport plugins such as *RTI TCP Transport*[1] are described in their respective chapters in this manual.

For example:

```
<participant_qos>
    <property>
      <value>
         <element>
             <name>dds.transport.UDPv4.builtin.parent.message_size_max
             </name>
             <value>65536</value>
         </element>
```

1. *RTI TCP Transport* is included with *RTI Data Distribution Service*, but is not enabled by default.

```
            <element>
                <name>dds.transport.UDPv4.builtin.send_socket_buffer_size
                </name>
                <value>131072</value>
            </element>
            <element>
                <name>dds.transport.UDPv4.builtin.recv_socket_buffer_size
                </name>
                <value>262144</value>
            </element>
        </value>
    </property>
</participant_qos>
```

15.7.7 Thread Settings

When thread priorities are configured using XML, the values are considered native priorities. For 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.

15.8 QoS Profiles

A QoS *profile* groups a set of related QoS, usually one per entity, identified by a name. For example:

```
<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 QoS-Profile Inheritance (Section 15.8.2).

There are functions that allow you to create DDS entities using profiles, such as **create_participant_with_profile()** (Section 8.3.1), **create_topic_with_profile()** (Section 5.1.1), etc.

If you create an entity using a profile without a QoS definition or an inherited QoS definition (see QoS-Profile Inheritance (Section 15.8.2)) for that class of entity, *RTI Data Distribution Service* uses the default QoS.

Example 1:

```
<qos_profile name="BatchStrictReliableCommunicationProfile"
base_name="StrictReliableCommunicationProfile">
    <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:

```
<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 *RTI Data Distribution Service* QoS.

15.8.1 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:

```
<datawriter_qos name="KeepAllWriter">
     <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
     </history>
</datawriter_qos>
```

is equivalent to:

```
<qos_profile name="KeepAllWriter">
     <datawriter_qos>
        <history>
           <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
     </datawriter_qos>
</qos_profile>
```

15.8.2 QoS-Profile Inheritance

An individual QoS or profile can inherit values from other QoSs or profiles described in the XML file by using the attribute, **base_name**.

Inheriting from other XML Files: A QoS or QoS Profile may inherit values from other QoSs or QoS Profiles described in different XML files. A QoS or profile can only inherit from other QoS policies or profiles that have already been loaded. The order in which XML resources are loaded is described in Section 15.2.

The following examples show how to inherit from other profiles:

Example 1:

```
<qos_library name="Library">
    <qos_profile name="BaseProfile">
        <datawriter_qos>
           ...
        </datawriter_qos>
        <datareader_qos>
           ...
        </datareader_qos>
    </qos_profile>

    <qos_profile name="DerivedProfile" base_name="BaseProfile">
        <datawriter_qos>
```

```
            ...
        </datawriter_qos>
        <datareader_qos>
            ...
        </datareader_qos>
    </qos_profile>
</qos_library>
```

The **writer_qos** and **reader_qos** in DerivedProfile inherit their values from the corresponding QoS in BaseProfile.

Example 2:

```
<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 profile definition with a single QoS (see Section 15.8.1).

Example 3:

```
<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.

Example 4:

```
<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></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.

Example 5:

```
<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.

15.8.3 Topic Filters

A QoS profile may contain several writer, reader and topic QoSs. *RTI Data Distribution Service* will select a QoS based on the evaluation of a filter expression 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*">
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
   </datawriter_qos>
```

```
    <datawriter_qos topic_filter="B*">
        <history>
            <kind>KEEP_ALL_HISTORY_QOS</kind>
        </history>
        <reliability>
            <kind>RELIABLE_RELIABILITY_QOS</kind>
        </reliability>
        <resource_limits>
            <max_samples>128</max_samples>
            <max_samples_per_instance>128</max_samples_per_instance>
            <initial_samples>128</initial_samples>
            <max_instances>1</max_instances>
            <initial_instances>1</initial_instances>
        </resource_limits>
    </datawriter_qos>
    …
</qos_profile>
```

If **topic_filter** is not specified in a QoS, *RTI Data Distribution Service* 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 Section 8.2.5)

DomainParticipant:

❑ **create_datawriter_with_profile()** (see Creating DataWriters (Section 6.3.1))

❑ **create_datareader_with_profile()** (see Creating DataReaders (Section 7.3.1)

❑ **create_topic_with_profile()** (see Creating Topics (Section 5.1.1))

Publisher:

❑ **create_datawriter_with_profile()** (see Creating DataWriters (Section 6.3.1))

Subscriber:

❑ **create_datareader_with_profile()** (see Creating DataReaders (Section 7.3.1))

Topic:

❑ **set_qos_with_profile()** (see Setting Topic QosPolicies (Section 5.1.3))

DataWriter:

- ❑ **set_qos_with_profile()** (see Changing QoS Settings After the Publisher Has Been Created (Section 6.2.4.2))

DataReader:

- ❑ **set_qos_with_profile()** (see Setting DataReader QosPolicies (Section 7.3.7))

Other APIs will ignore QoSs with a **topic_filter** value different than "*". A QoS Profile with QoSs using **topic_filter** can also inherits from other QoS Profiles. In this case, inheritance will consider the value of the **topic_filter** expression.

Example 1:

```
<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:

```
<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>
        <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:

```
<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.

15.8.4 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

In the following example, the *DataWriter* and *DataReader* default QoS will be overwritten with the values specified in a profile named '**StrictReliableCommunicationProfile**':

```
<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 **StrictReliable-CommunicationProfile**.

```
<qos_profile name="BestEffortCommunicationProfile"
    is_default_qos="true">
...
</qos_profile>
```

```
<qos_profile name="StrictReliableCommunicationProfile"
    is_default_qos="true">
...
</qos_profile>
```

15.8.5 Get Qos Profiles

To get a list of loaded 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)
```

15.9 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:

```
<dds>
    <qos_library name="RTILibrary">

      <!-- Individual QoSs are shortcuts for QoS Profiles with 1 QoS -->
        <datawriter_qos name="KeepAllWriter">
            <history>
                <kind>KEEP_ALL_HISTORY_QOS</kind>
            </history>
        </datawriter_qos>

        <!-- Qos Profile -->
        <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>

    </qos_library>
</dds>
```

A QoS library can be reopened within the same configuration file or across different configuration files. For example:

```
<dds>
    <qos_library name="RTILibrary">
        ...
     </qos_library>
     ...
    <qos_library name="RTILibrary">
        ...
     </qos_library>
</dds>
```

15.9.1 Get Qos Profile 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 *RTI Data Distribution Service*.

```
DDS_ReturnCode_t get_qos_profile_libraries (
                             struct DDS_StringSeq *profile_names)
```

15.10 URL Groups

To provide redundancy and fault tolerance, you can specify multiple locations for a single XML document via URL groups. The syntax of a URL group is:

```
[URL1 | URL2 | URL2 | ... | URLn]
```

For example:

```
[file:///usr/local/default_dds.xml | file:///usr/local/
alternative_default_dds.xml]
```

Only one of the elements in the group will be loaded by *RTI Data Distribution Service*, starting from the left.

Brackets are not required for groups with a single URL.

The **NDDS_QOS_PROFILES** environment variable contains a set of URL groups separated by semicolons. For example, on Linux and Solaris systems:

```
setenv NDDS_QOS_PROFILES [file:///usr/local/default_dds.xml|file:///
usr/local/alternative_default_dds.xml];[str://"<dds><qos_library
name="MyQosLibrary"></qos_library></dds>"]
```

The **url_profile** field in the PROFILE QosPolicy (DDS Extension) (Section 8.4.1) will contain a sequence of URL groups.

Chapter 16 Multi-channel DataWriters

In *RTI Data Distribution Service*, 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.

- **Consumer-based** approaches push the burden of filtering to the consumer side. The producer sends all the data to every consumer and the middleware on the consumer side decides whether the application wants it or not, automatically filtering the unwanted data. This approach is simple and fits well in systems that use multicast protocols as a transport. But the approach is not efficient for consumers that want small subsets of the data, since the consumers have to spend a lot of time filtering unwanted data. This approach is also unsuitable for systems with large volumes of data, such as the above Market Data system.
- **Server-based** approaches push the burden of filtering to a third component: a server or broker. This approach has some scalability advantages—the server can be run on a more powerful computer and can be federated to handle a large number of consumers. Some providers also provide hardware-assisted filtering in the server. However, the server-based approach significantly increases latency and jitter. It is also far more expensive to deploy and manage.
- **Network Switch-based** approaches leverage the network hardware, specifically advanced (IGMP snooping) network switches, to offload most of the burden of filtering from the producers and consumers without introducing additional hardware, servers or proxies. This approach preserves the low latency and ease of deployment of the brokerless approaches while still providing most of the offloading and scalability benefits of the broker.

RTI supports the producer-based, consumer-based and network-switch approaches to content filtering:

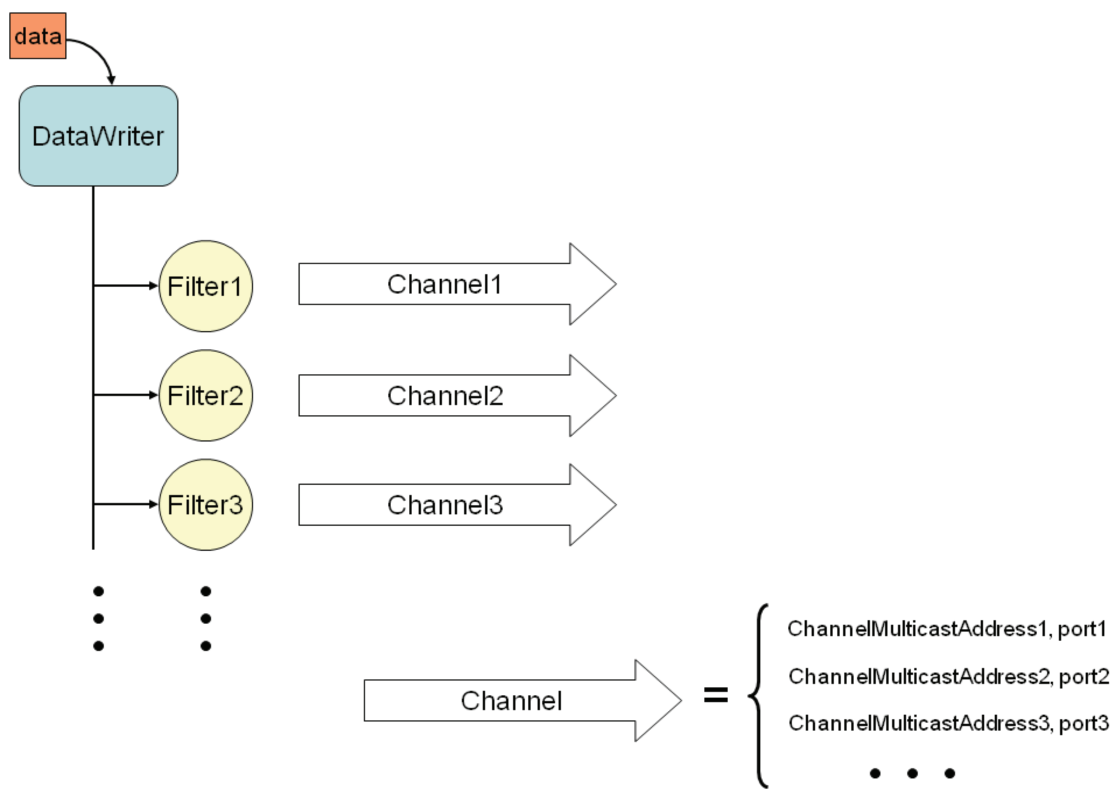
- RTI automatically uses the producer-based and consumer-based approaches as soon as it detects a consumer that specifies a content filter. The producer-based approach is used if the consumer is receiving data over a point-to-point protocol (i.e., not multicast) and the number of consumers that specify filters is reasonably low (below 32). Otherwise, RTI uses a subscriber-based approach.
- To use the more scalable network-switched based approach, an application must configure the *DataWriter* as a *Multi-channel DataWriter*. This concept is described in the following section.

16.1 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.

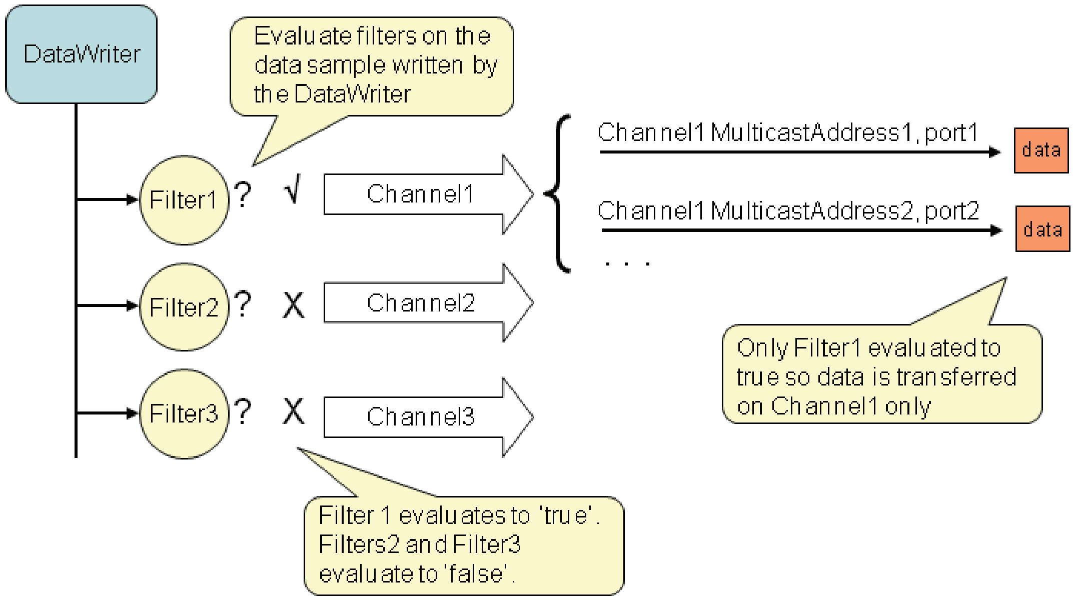
To determine which multicast addresses will be used to send the data, the middleware evaluates a set of filters that are configured for the *DataWriter*. Each filter "guards" a *channel*—a set of multicast addresses. Each time a multi-channel *DataWriter* writes data, the filters are applied. If a filter 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 16.1 **Multi-channel Data Flow**



16. Multi-channel DataWriters

Figure 16.2 **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.

16.2 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* MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12).

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:

If the Data Matches this Filter...	Send the Data to these Multicast Destinations
`Symbol MATCH '[A-K]*`	UDPv4:225.0.0.1:9000
`Symbol MATCH '[L-Q]*`	UDPv4:225.0.0.2:9001
`Symbol MATCH '[P-Z]*`	UDPv4:225.0.0.3:9002; 225.0.0.4:9003;

The example C++ code in Figure 16.3 on page 6 shows how to configure the channels.

The MULTI_CHANNEL QosPolicy is propagated along with discovery traffic. The value of this policy is available in the builtin topic for the publication (see the **locator_filter** field in Table 14.2, "Publication Built-in Topic's Data Type (DDS_PublicationBuiltinTopicData)," on page 14-3).

16.2.1 Limitations

When considering use of a multi-channel DataWriter, please be aware of the following limitations:

- ❏ A *DataWriter* that uses the MULTI_CHANNEL QosPolicy will ignore multicast and unicast addresses specified on the reader side through the TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5) and TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21). The *DataWriter* will not publish samples on these locators.

Figure 16.3 **Using the MULTI_CHANNEL QosPolicy**

```
// 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
writer = publisher->create_datawriter(
                     topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);
```

- ❑ Multi-channel *DataWriters* cannot be configured to use the Durable Writer History feature (described in Section 11.3).
- ❑ Multi-channel *DataWriters* do not support fragmentation of large data.
- ❑ Multi-channel *DataWriters* cannot be configured for asynchronous publishing (described in Section 6.4.1).

- Multi-channel *DataWriters* rely on the **rtps_object_id** in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2) to be DDS_RTPS_AUTO_ID (which causes automatic assignment of object IDs to channels).
- To guarantee reliable delivery, a *DataReader's* PRESENTATION QosPolicy (Section 6.4.6) 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 *DataWriter's* MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12) must use expressions that only refer to key fields.

16.3 Multi-channel Configuration on the Reader Side

No special changes are required in a subscribing application to get data from a multi-channel *DataWriter*.

If you want the *DataReader* to subscribe to only a subset of the channels, use a ContentFilteredTopic, as described in Section 5.4. For example:

```
// Create a content filtered topic
contentFilter = participant->create_contentfilteredtopic_with_filter(
                    "FilteredTopic",
                    topic,
                    "symbol MATCH 'NYE/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, *RTI Data Distribution Service* takes care of all the necessary steps:

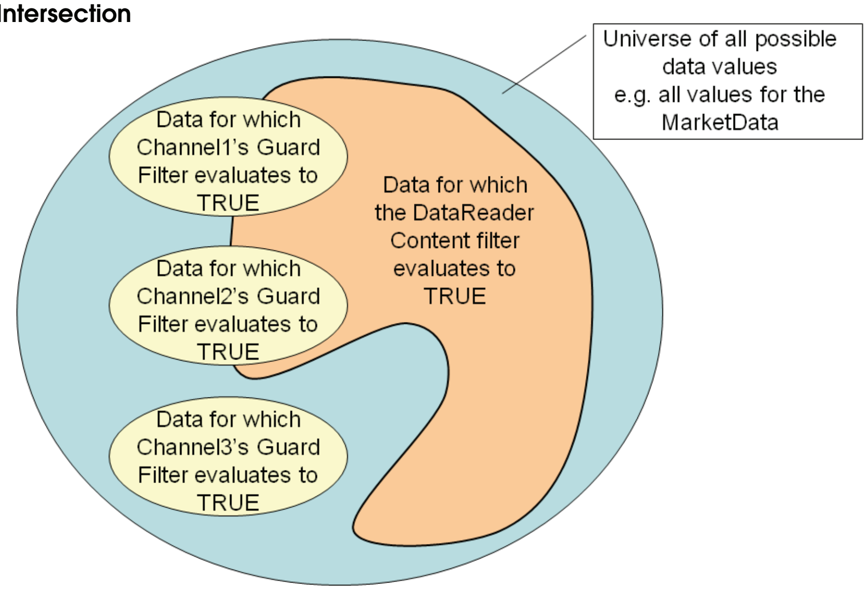
- The *DataReader* automatically discovers all the *DataWriters*—including multi-channel *DataWriters*—for the Topic it subscribes to.
- When the *DataReader* discovers a multi-channel *DataWriter*, it also discovers the list of channels used by that *DataWriter*.

- When the multi-channel *DataWriter* discovers a *DataReader*, it also discovers the content filters specified by that *DataReader*, if any.

With all this information, *RTI Data Distribution Service* 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 16.4 **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.

16.4 Where Does the Filtering Occur?

If Multi-channel *DataWriters* are used, the filtering can occur in three places:

- Filtering at the DataWriter (Section 16.4.1)
- Filtering at the DataReader (Section 16.4.2)
- Filtering on the Network Hardware (Section 16.4.3)

16.4.1 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 Section 16.7, 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.

16.4.2 Filtering at the DataReader

The *DataReader* will listen on the multicast addresses that correspond to the channels of interest (see Section 16.3). 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 samples corresponding to stock symbols such as 'MSFT' and 'APPL'. The *DataReader* must filter these samples out.

As explained in Section 16.7, 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.

16.4.3 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 Section 16.7.1).

16.5 Fault Tolerance and Redundancy

To achieve fault tolerance and redundancy, configure the *DataWriter's* MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12) to publish a sample over multiple channels or over different multicast addresses within a single channel. Figure 16.5 shows how to use overlapping channels.

If a sample is published to multiple multicast addresses, a *DataReader* may receive multiple copies of the 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 Section 11.4.2).

Figure 16.5 **Using the MULTI_CHANNEL QosPolicy with Overlapping Channels**

```
// 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
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(2,2);
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 '[C-Z]*'");

// Symbols starting with [C-M] will be published in two different channels

// Create writer
writer = publisher->create_datawriter(
    topic, writer_qos, NULL, DDS_STATUS_MASK_NONE);
```

16.6 Reliability with Multi-Channel DataWriters

16.6.1 Reliable Delivery

Reliable delivery is only guaranteed when the **access_scope** in the *Subscriber's* PRESENTATION QosPolicy (Section 6.4.6) is set to DDS_INSTANCE_PRESENTATION_QOS (default value) and the filters in the *DataWriter's* MULTI_CHANNEL QosPolicy (DDS Extension) (Section 6.5.12)) are keyed-only based.

Market Data Example, continued:

Given the following IDL description for our MarketData topic type:

```
Struct MarketData {
    string<255> Symbol; //@key
    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, *RTI Data Distribution Service* only guarantees reception of the most recent data from the multi-Channel *DataWriter*.

16.6.2 Reliable Protocol Considerations

Reliability is maintained on a per-channel basis. Each channel has its own reliability channel send queue. The size of that queue is limited by **max_samples** in the RESOURCE_LIMITS QosPolicy (Section 6.5.18) and/or **max_batches** in DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3). The protocol parameters described in the DATA_WRITER_PROTOCOL QosPolicy (DDS Extension) (Section 6.5.2) are applied per channel, with the following exceptions:

❑ **low_watermark** and **high_watermark**: The low watermark and high watermark control the queue levels (in number of samples) that determine when to switch between regular and fast heartbeat rates (see Section 6.5.2.1). With multi-channel

DataWriters, **high_watermark** and **low_watermark** refer to the *DataWriter*'s queue (not the reliability channel queue). Therefore, periodic heartbeating cannot be controlled on a per-channel basis.

Important: With multi-channel *DataWriters*, **low_watermark** and **high_watermark** refer to application samples even if batching is enabled. This behavior differs from the one without multi-channel *DataWriters* (where **low_watermark** and **high_watermark** refer to batches).

- **heartbeats_per_max_samples**: This field defines the number of heartbeats per send queue. For multi-channel *DataWriters*, the value is applied per channel. However, the send queue size that is used to calculate the a piggyback heartbeat rate is defined per *DataWriter* (see **max_samples** in RESOURCE_LIMITS QosPolicy (Section 6.5.18)).

  Important: With multi-channel *DataWriters*, **heartbeats_per_max_samples** refers to samples even if batching is enabled. This behavior differs from the one without multi-channel *DataWriters* (where **heartbeats_per_max_samples** refers to batches, Section 6.5.2).

With batching and multi-channel *DataWriters*, the size of the *DataWriter's* send queue should be configured using **max_samples** (in RESOURCE_LIMITS QosPolicy (Section 6.5.18)) instead of **max_batches** (in DATA_WRITER_RESOURCE_LIMITS QosPolicy (DDS Extension) (Section 6.5.3)) in order to take advantage of **heartbeats_per_max_samples**.

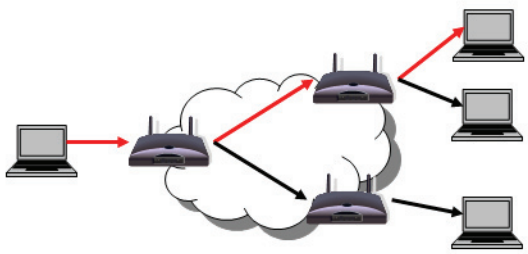
16.7 Performance Considerations

16.7.1 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 16.6.

Figure 16.6 **IGMP Snooping**



16.7.2 DataWriter and DataReader Filtering

Where Does the Filtering Occur? (Section 16.4) 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 Section 2.2.2). 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:

```
struct {
    string<MAX_SYMBOL_SIZE> Symbol; //@key
}
```

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 RESOURCE_LIMITS QosPolicy (Section 6.5.18) on both the writer and reader sides. A higher number of buckets will provide better performance, but requires more resources.

Chapter 17 RTI Data Distribution Service Threading Model

This chapter describes the internal threads that *RTI Data Distribution Service* uses for sending and receiving data, maintaining internal state, and calling user code when events occur such as the arrival of new data samples. It may be important for you to understand how these threads may interact with your application.

An *RTI Data Distribution Service DomainParticipant* uses three types of threads. The actual number of threads depends on the configuration of various QosPolicies as well as the implementation of the transports used by the *DomainParticipant* to send and receive data.

- ❏ Database Thread (Section 17.1)
- ❏ Event Thread (Section 17.2)
- ❏ Receive Threads (Section 17.3)
- ❏ Exclusive Areas, RTI Data Distribution Service Threads and User Listeners (Section 17.4)
- ❏ Controlling CPU Core Affinity for RTI Threads (Section 17.5)

Through various QosPolicies, the user application can configure the priorities and other properties of the threads created by *RTI Data Distribution Service*. 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.

17.1 Database Thread

RTI Data Distribution Service uses internal data structures to store information about locally-created and remotely-discovered DDS *Entities*. In addition, it will store various objects and data used by *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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*.

The DATABASE QosPolicy (DDS Extension) (Section 8.5.1) 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.

17.2 Event Thread

During operation, *RTI Data Distribution Service* 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 DEADLINE QosPolicy (Section 6.5.4) is used to ensure that *DataWriters* have published data or *DataReaders* have received data within a specified time period. Similarly, the LIVELINESS QosPolicy (Section 6.5.11) configures *RTI Data Distribution Service* 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 Chapter 10: Reliable Communications.

The checking of whether or not deadlines have been missed, the invoking of user-installed *Listener* callbacks to notify the application missed deadlines, and the sending of heartbeats to maintain reliable connections are all done with an internal *RTI Data Distribution Service* thread called the Event thread.

❑ Only one Event thread is created per *DomainParticipant*.

The EVENT QosPolicy (DDS Extension) (Section 8.5.6) 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.

17.3 Receive Threads

RTI Data Distribution Service 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 data-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 Section 7.3.3).

The number of Receive threads that *RTI Data Distribution Service* 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.

RTI Data Distribution Service will try to create receive resources[1] for every port of every transport on which it is configured to receive messages. The TRANSPORT_UNICAST QosPolicy (DDS Extension) (Section 6.5.21) for *DomainParticipant*, *DataWriters*, and *DataReaders*, the TRANSPORT_MULTICAST QosPolicy (DDS Extension) (Section 7.6.5) for *DataReaders* and the DISCOVERY QosPolicy (DDS Extension) (Section 8.5.2) for *DomainParticipants* all configure the number of ports and the number of transports that *RTI Data Distribution Service* will try to use for receiving messages.

Generally, transports will require *RTI Data Distribution Service* 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 *RTI Data Distribution Service*. Sometimes *RTI Data Distribution Service* only needs to create a single receive resource for any number of ports.

When *RTI Data Distribution Service* 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 *RTI Data Distribution Service* will not have to create a new receive resource. If not, then *RTI Data Distribution Service* 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? *RTI Data Distribution Service* will create a Receive thread to service every receive resource that is created. If you use a socket analogy, then for every socket created, *RTI Data Distribution Service* will use a separate thread to process the data received on that socket.

So how many thread will *RTI Data Distribution Service* create by default–using only the builtin UDPv4 and shared memory transports and without modifying any QosPolicies?

Three Receive threads are created for meta-traffic[1]:

❏ 2 for unicast (one for UDPv4, one for shared memory)

❏ 1 for multicast (for UDPv4)[2]

Three Receive threads created for user data:

❏ 2 for unicast (UDPv4, shared memory)

❏ 1 for multicast (UDPv4)

1. If UDPv4 was the only transport that *RTI Data Distribution Service* supports, then we would have called these receive resources, sockets.

1. Meta-traffic refers to traffic internal to *RTI Data Distribution Service* related to dynamic discovery (see Chapter 12: Discovery).

2. Multicast is not supported by shared memory transports.

Therefore, by default, you will have a total of 6 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 RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7). However, note that the thread properties in the RECEIVER_POOL QosPolicy apply to all Receive threads created for the *DomainParticipant*.

17.4 Exclusive Areas, RTI Data Distribution Service Threads and User Listeners

RTI Data Distribution Service Event and Receive threads may invoke user code through the *Listener* callbacks installed on different *Entities* while executing internal *RTI Data Distribution Service* code. In turn, user code inside the callbacks may invoke *RTI Data Distribution Service* APIs that reenter the internal code space of *RTI Data Distribution Service*. For thread safety, *RTI Data Distribution Service* allocates and uses mutual exclusion semaphores (mutexes).

As discussed in Section 4.5, when multiple threads and multiple mutexes are mixed together, deadlock may result. To prevent deadlock from occurring, *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* code. However, the more EAs that exist, the more restrictions on the *RTI Data Distribution Service* APIs that are allowed to be invoked in *Entity Listener* callbacks.

The EXCLUSIVE_AREA QosPolicy (DDS Extension) (Section 6.4.3) control how many EAs will be created by *RTI Data Distribution Service*. For a more detailed discussion on EAs and the restrictions on the use of *RTI Data Distribution Service* APIs within *Entity Listener* methods, please see Exclusive Areas (EAs) (Section 4.5).

17.5 Controlling CPU Core Affinity for RTI Threads

Two fields in the DDS_ThreadSettings_t structure 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 RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7).

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 **cpu_list** ={0,1} and that the middleware creates three receive threads: {A, B, C}. If **cpu_rotation** is set to 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 **cpu_rotation** is CPU_RR_ROTATION, each thread will be assigned in round-robin fashion to one of the processors in **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 Platform Notes describe which architectures support this feature.

Chapter 18 Troubleshooting

This chapter contains tips on troubleshooting *RTI Data Distribution Service* applications. For an up-to-date list of frequently asked questions, see the RTI Customer Portal, accessible from www.rti.com/support—select the **Find Solution** link to see sample code, general information on *RTI Data Distribution Service*, performance information, troubleshooting tips, and technical details.

This chapter contains the following sections:

- ❑ What Version am I Running? (Section 18.1)
- ❑ Controlling Messages from RTI Data Distribution Service (Section 18.2)

18.1 What Version am I Running?

There are two ways to obtain version information:

- ❑ By looking at the revision files, as described in Section 18.1.1.
- ❑ Programmatically at run time, as described in Section 18.1.2.

18.1.1 Finding Version Information in Revision Files

In the top-level directory of your *RTI Data Distribution Service* 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.4.5c** and **rev_persistence_rtidds4.5c**. Each file contains more details, such as a patch level and if the product is license managed.

18. Troubleshooting

For example:

```
Host Build 4.5c rev 04 (0x04050200)
```

The revision files for *RTI Data Distribution Service* target libraries are in the same directory as the libraries (**${NDDSHOME}/lib/<*architecture*>**).

18.1.2 Finding Version Information Programmatically

The methods in the NDDSConfigVersion class can be used to retrieve version information for the *RTI Data Distribution Service* 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 letter release
- ❑ A build number

Table 18.4 lists the available operations (they will vary somewhat depending on the programming language you are using; consult the online documentation for more information).

Table 18.1 **NDDSConfigVersion Operations**

Purpose	Operation	Description
To retrieve version information in a structured format	get_product_version	Gets version information for the *RTI Data Distribution Service* product.
	get_core_version	Gets version information for the *RTI Data Distribution Service* core library.
	get_c_api_version	Gets version information for the *RTI Data Distribution Service* C library.
	get_cpp_api_version	Gets version information for the *RTI Data Distribution Service* C++ library.
To retrieve version information in string format	to_string	Converts the version information for each library into a string. The strings for each library are put in a single hyphen-delimited list.

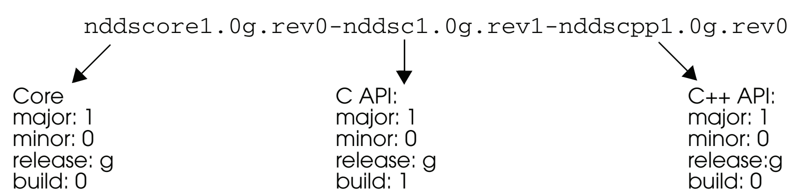
The **get_product_version()** operation returns a reference to a structure of type DDS_ProductVersion_t:

```
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:

```
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 *RTI Data Distribution Service* core, followed by the C and C++ API libraries, separated by hyphens. For example:



18.2 Controlling Messages from RTI Data Distribution Service

RTI Data Distribution Service 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.

How much information is logged is known as the *verbosity* setting. Table 18.2 describes the six increasing verbosity levels.

Table 18.2 **Message Logging Verbosity Levels**

Verbosity (NDDS_CONFIG_ LOG_VERBOSITY_*)	Description
SILENT	No messages will be logged. (lowest verbosity)
ERROR (default level for all categories)	Log only high-priority error messages. An error indicates something is wrong with how *RTI Data Distribution Service* is functioning. The most common cause of this type of error is an incorrect configuration.
WARNING	Additionally log warning messages. A warning indicates that *RTI Data Distribution Service* 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.
STATUS_LOCAL	Additionally log verbose information about the lifecycles of local *RTI Data Distribution Service* objects.
STATUS_REMOTE	Additionally log verbose information about the lifecycles of remote *RTI Data Distribution Service* objects.
STATUS_ALL	Additionally log verbose information about periodic activities and *RTI Data Distribution Service* threads. (highest verbosity)

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**.

Caution: 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.

You will typically change the verbosity of all of *RTI Data Distribution Service* 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 *RTI Data Distribution Service* fall into the five categories listed in Table 18.3; each category can be set to a different verbosity level.

Table 18.3 **Message Logging Categories**

Category (NDDS_CONFIG_ LOG_CATEGORY_*)	Description
PLATFORM	Messages about the underlying platform (hardware and OS).
COMMUNICATION	Messages about data serialization and deserialization and network traffic.
DATABASE	Messages about the internal database of *RTI Data Distribution Service* objects.
ENTITIES	Messages about local and remote entities and the discovery process.
API	Messages about *RTI Data Distribution Service*'s API layer (such as method argument validation).

The methods in the **NDDSConfigLogger** class can be used to change verbosity settings, as well as the destination for logged messages. Table 18.4 lists the available operations; consult the online documentation for more information.

Table 18.4 **NDDSConfigLogger Operations**

Purpose	Operation	Description
Change Verbosity for all Categories	get_verbosity	Gets the current verbosity. If per-category verbosities are used, returns the highest verbosity of any category.
	set_verbosity	Sets the verbosity of all categories.
Change Verbosity for a Specific Category	get_verbosity_by_category	Gets/Sets the verbosity for a specific category.
	set_verbosity_by_category	
Change Destination of Logged Messages	get_output_file	Returns the file to which messages are being logged, or NULL for the default destination (standard output on most platforms).
	set_output_file	Redirects future logged messages to the specified file (or NULL to return to the default).
Change Message Format	get_print_format	Gets/Sets the current message format that *RTI Data Distribution Service* is using to log diagnostic information. See Format of Logged Messages (Section 18.2.1).
	set_print_format	

18.2.1 Format of Logged Messages

You can control the amount of information in each message with the **set_print_format()** operation. The format options are listed in Step 18.5.

Table 18.5 **Message Formats**

Message Format (NDDS_CONFIG_LOG_ PRINT_FORMAT_*)	Description
DEFAULT	Message, method name, and activity context.
TIMESTAMPED	Message, method name, activity context, and timestamp.
VERBOSE	Message with all available context information (includes thread identifier, activity context).
VERBOSE_TIMESTAMPED	Message with all available context information and timestamp.
DEBUG	Information for internal debugging by RTI personnel.
MINIMAL	Message number, method name.
MAXIMAL	All available fields.

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 *RTI Data Distribution Service*. 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"in the second line of the above stack trace means "cannot create socket."

18.2.1.1 Timestamps

Reported times are in seconds from a system-dependent starting time; these are equivalent to the output format from *RTI Data Distribution Service*. The timestamp is in the form "ssssss.mmmmmm" where <ssssss> is a number of seconds, and <mmmmm> is a fraction of a 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()**.

18.2.1.2 Thread identification

Thread identification strings uniquely identify for active thread when a message is output to the console. A thread may be a user (application) thread or one of several types of internal threads. The possible thread types are:

- ❏ user thread: U<threadID>
- ❏ receive thread: rR<thread index><domain ID><app ID>, where thread index is an integer identifying this receive thread
- ❏ event thread: revt<domain ID><app ID>
- ❏ asynchronous publisher thread: rDsp

Logging of thread IDs are not enabled by default. To enable it, use NDDS_Config_Logger method **set_print_format()**.

18.2.1.3 Hierarchical Context

Many middleware APIs now store information in thread-specific storage about the current operation, as well as information about which domain (and participant ID) was active, and which entities were being operated on. In the case of objects that are associated with topics, the topic name is also stored.

The context field is output by default.

18.2.1.4 Explanation of Context Strings

- ❏ Domain context

  ```
  Dxxyy
  ```

  In this case, xx = participant ID, yy = domain #. For example, **D0149** means "domain 49, participant 01."

- ❏ Entity context

  Operation on an entity will specify the object and a numeric ID, such as **Writer(001A1)**. The name will be one of the following:

String	Object type
Participant	DDS_DomainParticipant
Pub	DDS_Publisher
Sub	DDS_Subscriber
Topic	DDS_Topic

Writer	DDS_<*>DataWriter
Reader	DDS_<*>DataReader

- ❑ Topic Context

  T=Hello refers to topic "Hello."

  The operations which report context include:

String	Operation
Entity operations	
ENABLE	Entity::enable
GET_QOS	Entity::get_qos
SET_QOS	Entity::set_qos
GET_LISTENER	Entity::get_listener
SET_LISTENER	Entity::set_listener
Factory operations (DP Factory, Participant, Pub/Sub)	
CREATE <Entity>	Factory::create_<entity>
DELETE <Entity>	Factory::delete_<entity>
GET_DEFAULT_QOS <Entity>	Factory::get_default_<entity>_qos
SET_DEFAULT_QOS <Entity>	Factory::set_default_<entity>_qos
Participant-specific operations	
GET_PUBS	Participant::get_publishers
GET_SUBS	Participant::get_subscribers
LOOKUP Topic(<name>)	Participant::lookup_topicdescription
LOOKUP FlowController(<name>)	Participant::lookup_flowcontroller
IGNORE <Entity>(<host ID>)	Participant::ignore_<entity>

Part 4: RTI Secure WAN Transport

The material in this part of the manual is only relevant if you have installed *RTI Secure WAN Transport*.

This feature is not part of the standard *RTI Data Distribution Service* 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 Secure WAN Transport is an optional package that enables participant discovery and data exchange in a secure manner over the public WAN. *RTI Secure WAN Transport* enables *RTI Data Distribution Service* 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.

❏ Chapter 19: RTI Secure WAN Transport

❏ Chapter 20: Configuring RTI Secure WAN Transport

Chapter 19 RTI Secure WAN Transport

RTI Secure WAN Transport provides transport plugins that can be used by developers of *RTI Data Distribution Service* applications. These transport plugins allow *RTI Data Distribution Service* applications running on private networks to communicate securely over a Wide-Area Network (WAN), such 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 *RTI Data Distribution Service* core is transport-agnostic. *RTI Data Distribution Service* 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 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 DDS. *RTI 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.

RTI 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 [draft-ietfbehave-rfc3489bis], 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:

- ❏ WAN Traversal via UDP Hole-punching (Section 19.1)
- ❏ WAN Locators (Section 19.2)
- ❏ Datagram Transport-Layer Security (DTLS) (Section 19.3)
- ❏ Certificate Support (Section 19.4)

For information on how to use *RTI Secure WAN Transport* with your *RTI Data Distribution Service* application, see Chapter 20: Configuring RTI Secure WAN Transport.

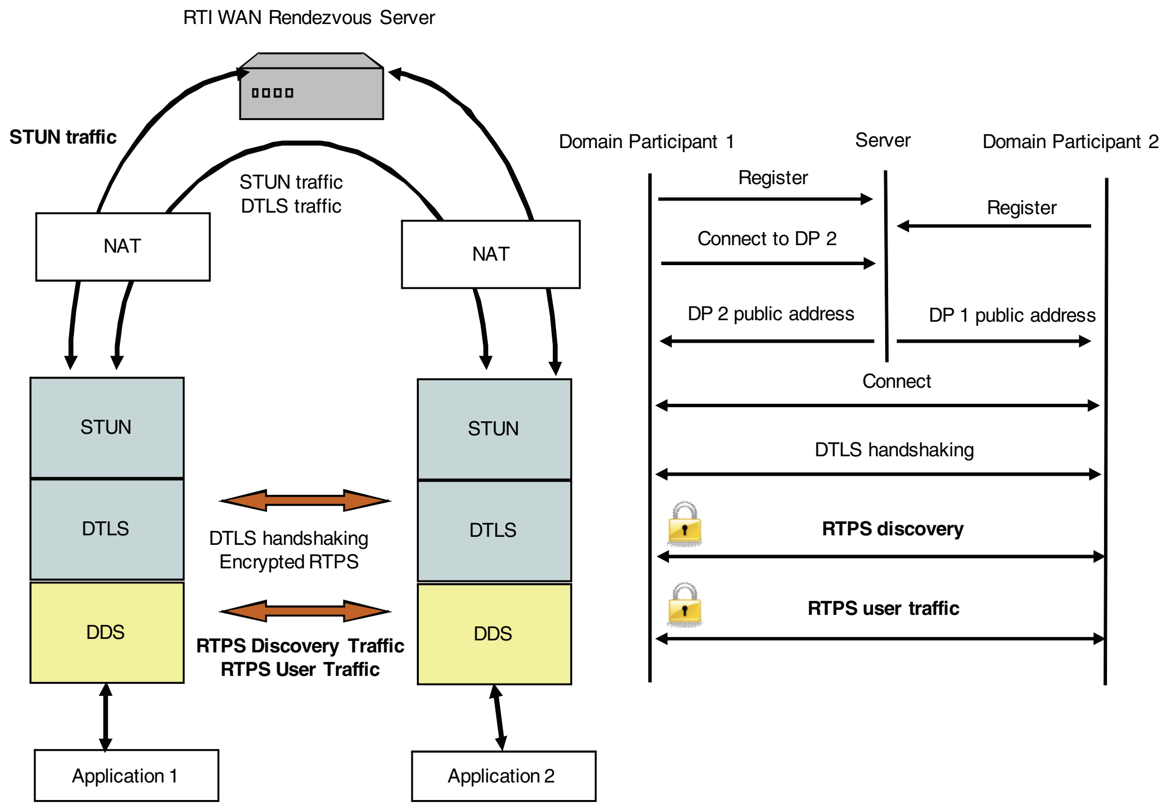
19.1 WAN Traversal via UDP Hole-punching

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 be 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 19.1 shows the RTI WAN transport architecture.

Figure 19.1 **RTI WAN Transport Architecture**



19.1.1 Protocol Details

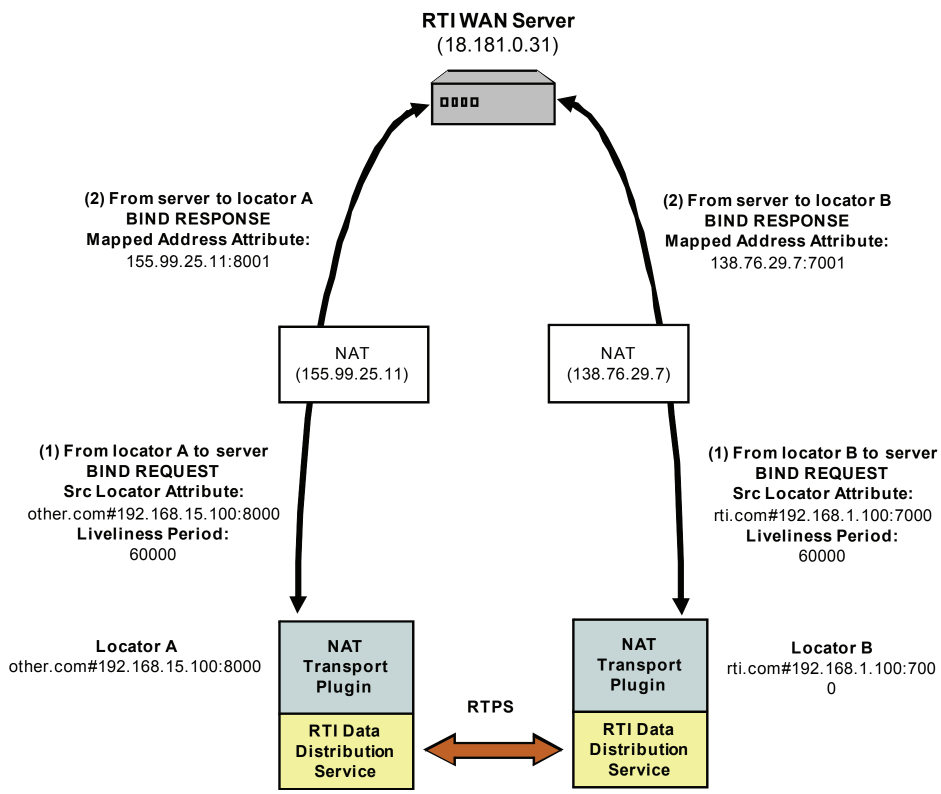
The UDP hole-punching algorithm implemented by the WAN transport has two different phases: registration and connection. This algorithm only works with cone or asymmetric NATs where the same public address/port is assigned to all the sessions with the same private address/port address.

- 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 19.2 illustrates the registration phase.

Figure 19.2 **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 *RTI Data Distribution Service* 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 19.3 on page 19-7) and the NAT hole punching process (Figure 19.4 on page 19-8).

- ❑ STUN Liveliness

  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.

Figure 19.3 **Connect Process**

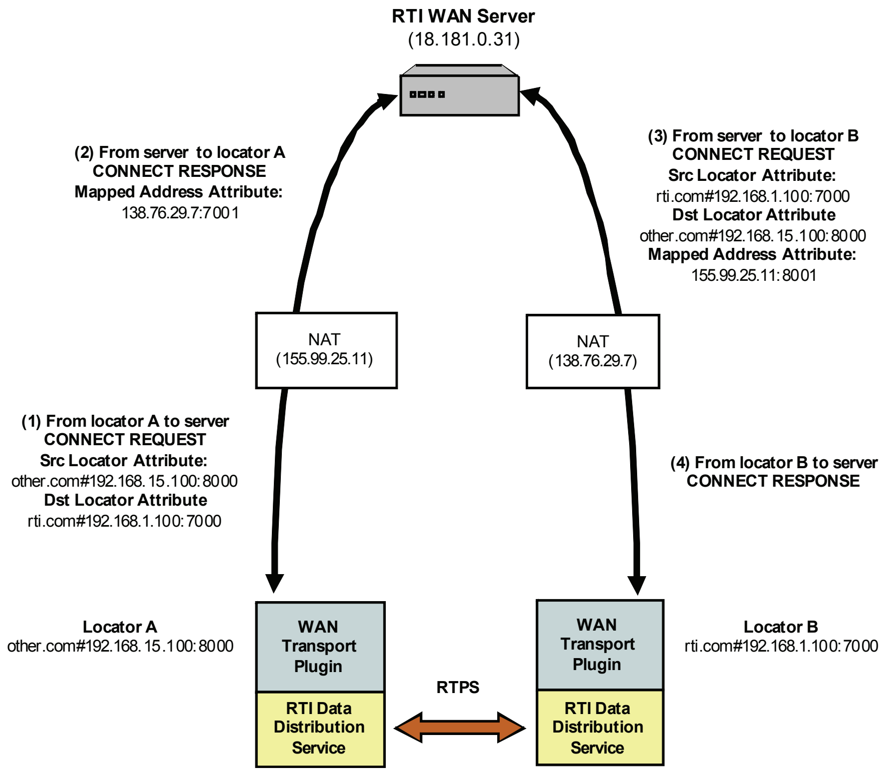
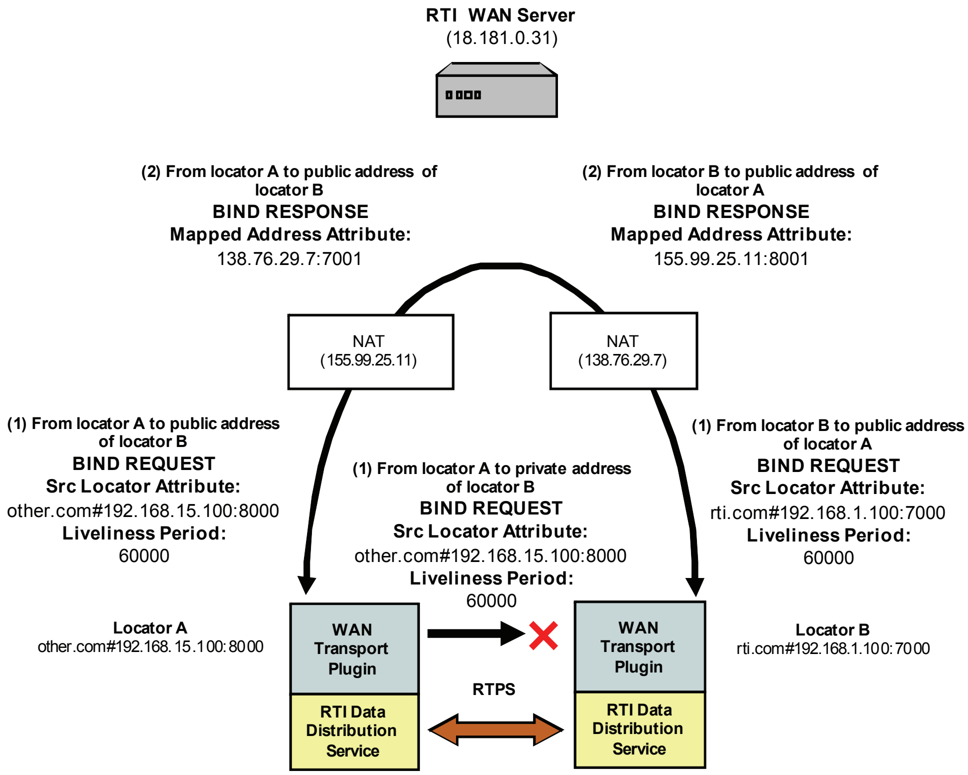
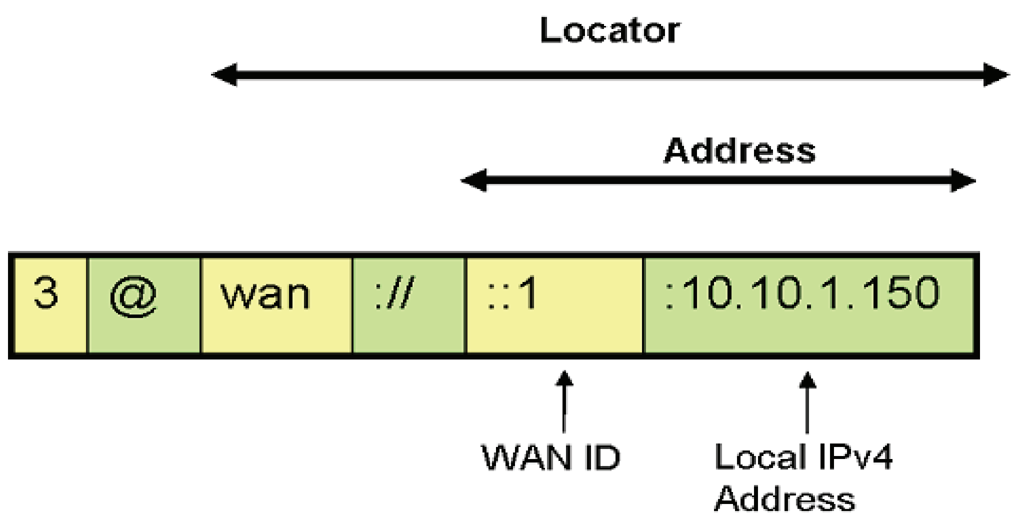


Figure 19.4 **NAT Hole Punching Process**



19.2 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:



- The address is a 128-bit address in IPv6 notation.
- 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 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::initial_peers QoS with the WAN locator addresses corresponding to the peers to which it wants to connect to. The value of DDS_DiscoveryQosPolicy::initial_peers can be set using the environment variable NDDS_DISCOVERY_PEERS or the NDDS_DISCOVERY_PEERS configuration file. (See Configuring the Peers List Used in Discovery (Section 12.2).)

19.3 Datagram Transport-Layer Security (DTLS)

Data security is provided by wrapping all 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 19.1 on page 19-4.

19.3.1 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 DDS is that even though DTLS is a connectionless protocol, it still has client/server semantics. The RTI Secure Transport maps a bidirectional 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).

19.3.2 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.

19.4 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) or the book, *"Network Security with OpenSSL"* by Viega, Messier, & Chandra (O'Reilly 2002; http://www.opensslbook.com), 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:

```
openssl ca -create serial -config openssl.cnf -days 365 \
    -in peer1req.pem -out myCA/newcerts/peer1cert.pem
```

d. Optionally, append the private key to the peer certificate:

```
cat myCA/newcerts/peer1cert.pem peer1key.pem \
     $>${private location}/ peer1.pem
```

19.5 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**.

```
* 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 aheared 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
* 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 20 Configuring RTI Secure WAN Transport

The *RTI 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 *RTI Data Distribution Service* 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 UNIX, Solaris and Windows systems only). This process is described in Setting Up a Transport with the Property QoS (Section 20.2).
- ❑ By instantiating a new transport (Section 20.5) and then registering it with the *DomainParticipant*, see Section 13.7 (not available in Java API).

Refer to the online documentation for details on these two approaches.

20.1 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

1. Security is disabled by default.

example is located in <RTI_Data_Distribution_Service_INSTALL_ROOT>/example/<language>/HelloWorldWAN.

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 Section 20.2. This will cause *RTI Data Distribution Service* 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 *RTI Data Distribution Service* (see Section 13.7). This way is *not* shown in the example. See Explicitly Instantiating a WAN or Secure Transport Plugin (Section 20.5).

20.2 Setting Up a Transport with the Property QoS

The PROPERTY QosPolicy (DDS Extension) (Section 6.5.15) allows you to set up name/value pairs of data and attach them to an entity, such as a *DomainParticipant*. This will cause *RTI Data Distribution Service* to dynamically load the WAN or Secure Transport libraries at run time and then implicitly create and register the transport plugin.

Please refer to Setting Builtin Transport Properties with the PropertyQosPolicy (Section 13.6).

To assign properties, use the **add_property()** operation:

```
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 6.48, "PropertyQoSPolicyHelper Operations," on page 6-139, as well as the online (HTML) documentation.

The 'name' part of the name/value pairs is a predefined string, described in WAN Transport Properties (Section 20.3) and Secure Transport Properties (Section 20.4).

Here are the basic steps, taken from the example Hello World application (for details, please see the example application.)

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);
```

Important! 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.

20.3 WAN Transport Properties

Table 20.1 lists the properties that you can set for the WAN Transport.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
dds.transport.load_plugins *(note: this does not take a prefix)*	***Required*** Comma-separated strings indicating the prefix names of all plugins that will be loaded by *RTI Data Distribution Service*. For example: "dds.transport.WAN.wan1". You will use this string as the prefix to the property names. See [a]. Note: you can load up to 8 plugins.
library	***Required*** Must set to "**libnddstransportwan.so**" (for UNIX/Solaris systems) or "**nddstransportwan.dll**" (for Windows system). This library and the dependent OpenSSL libraries need to be in the path during run time for use by *RTI Data Distribution Service* (in the LD_LIBRARY_PATH environment variable on UNIX/Solaris systems, in Path for Windows systems).
create_function	***Required*** Must be "**NDDS_Transport_WAN_create**"
aliases	Used to register the transport plugin returned by NDDS_Transport_WAN_create() (as specified by <WAN_prefix>.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[a] is used as the default alias for the plugin.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
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
parent.parent.address_bit_count	Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, the address_bit_count should be set to 8.
parent.parent.properties_bitmap	A bitmap that defines various properties of the transport to the *RTI Data Distribution Service* core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when *RTI Data Distribution Service* tries to receive a message using the plugin. This is in support of a zero-copy interface.
parent.parent.gather_send_ buffer_count_max	Specifies the maximum number of buffers that *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* from trying to gather too many buffers into a send call for the transport plugin. *RTI Data Distribution Service* requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is defined as NDDS_TRANSPORT_-PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
parent.parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plugin. This value must be set before the transport plugin is registered, so that *RTI Data Distribution Service* can properly use the plugin. If you set this higher than the default, the *DomainParticipant's* **buffer_size** (in the RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7)) should also be changed.
parent.parent.allow_interfaces	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.parent.deny_interfaces 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 *RTI Data Distribution Service* 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.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
parent.parent.deny_interfaces	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 "black" list is applied after the parent.parent.allow_interfaces 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 *RTI Data Distribution Service* 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.
parent.send_socket_buffer_size	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. The maximum value is operating system-dependent. If you configure this parameter to be NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT, then **setsockopt()** (or equivalent) will not be called to size the send buffer of the socket.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
parent.recv_socket_buffer_size	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. The maximum value is operating system-dependent. If set to NDDS_TRANSPORT_UDPV4_SOCKET_BUFFER_SIZE_OS_DEFAULT, then **setsockopt()** (or equivalent) will not be called to size the receive buffer of the socket.
parent.unicast_enabled	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.
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: Automatic. Lets *RTI Data Distribution Service* decide among the above two choices. If a shared memory transport plugin is available for local traffic, the effective value is 1 (i.e., disable UDPv4 local traffic). Otherwise, the effective value is 0, i.e., use UDPv4 for local traffic also.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
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.
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 *RTI Data Distribution Service* will use the zero-copy API if offered by the OS.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
parent.send_blocking	Controls the blocking behavior of send sockets. CHANGING THIS FROM THE DEFAULT CAN CAUSE SIGNIFICANT PERFORMANCE PROBLEMS. Currently two values are defined: NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS: Sockets are blocking (default socket options for Operating System). NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER: Sockets are modified to make them non-blocking. THIS IS NOT A SUPPORTED CONFIGURATION AND MAY CAUSE SIGNIFICANT PERFORMANCE PROBLEMS.
parent.transport_priority_mask	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.
parent.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 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.
parent.transport_priority_ mapping_high	
enable_security	***Required if you want to use security.***
recv_decode_buffer_size	Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data.
port_offset	Port offset to allow coexistence with non-secure UDP transport.
dtls_handshake_resend_interval	DTLS handshake retransmission interval in milliseconds

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
tls.verify.ca_file	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, ca_file or ca_path must be specified; both may be specified.***
tls.verify.ca_path	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. ***If you want to use security, ca_file or ca_path must be specified; both may be specified.***
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify.verify_peer	If non-zero, use mutual authentication when performing TLS handshake (default). If zero, only the reader side will present a certificate, which will be verified by the writer side.
tls.verify.verify_callback	This can be set to one of three values: -- "default" selects the default callback **NDDS_Transport_TLS_default_verify_callback()** -- "verbose" selects the verbose callback **NDDS_Transport_TLS_verbose_verify_callback()** -- "none" requests no callback be registered
tls.cipher.cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string.
tls.cipher.dh_param_files	List of available Diffie-Hellman (DH) key files. For example: "foo.h:512,bar.h:256" means: dh_param_files[0].file = foo.pem, dh_param_files[0].bits = 512, dh_param_files[1].file = bar.pem, dh_param_files[1].bits = 256
tls.cipher.engine_id	String ID of OpenSSL cipher engine to request.

Table 20.1 **Properties for NDDS_Transport_WAN_Property_t**

Property Name (prefix with 'dds.transport.WAN.wan1.')[a]	Property Value Description
tls.identity.certificate_chain_file	***Required if you want to use security.*** 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).
transport_instance_id[0] to [NDDS_TRANSPORT_ WAN_TRANSPORT_ INSTANCE_ID_LENGTH]	***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 "01,02" results in the array being set to: {0,0,0,0,0,0,0,0,0,0,1,2}
interface_address	Locator, as a string
server	***Required*** Server locator, as a string.
server_port	Server port number.
stun_retransmission_interval	STUN request messages requiring a response are resent with this interval. The interval is doubled after each retransmission. Specified in msec.
stun_number_of_retransmissions	Maximum number of times STUN messages are resent unless a response is received.
stun_liveliness_period	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.

a. 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.

20.4 Secure Transport Properties

Table 20.2 lists the properties that you can set for the Secure Transport.

Table 20.2 **Properties for NDDS_Transport_DTLS_Property_t**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[a]	Property Value Description
dds.transport.load_plugins *(note: this does not take a prefix)*	***Required*** Comma-separated strings indicating the prefix names of all plugins that will be loaded by *RTI Data Distribution Service*. For example: "dds.transport.DTLS.dtls1". You will use this string as the prefix to the property names. See [a]. Note: you can load up to 8 plugins.
library	***Required*** Must set to "libnddstransporttls.so" (for UNIX/Solaris) or "nddstransporttls.dll" (for Windows). This library and the dependent Openssl libraries need to be in the path during run time for use by *RTI Data Distribution Service* (in the LD_LIBRARY_PATH environment variable on UNIX/Solaris systems, in PATH for Windows systems).
create_function	***Required*** Must be "NDDS_Transport_DTLS_create"
aliases	Used to register the transport plugin returned by NDDS_Transport_DTLS_create() (as specified by <DTLS_prefix>.create_function) to the DomainParticipant. Aliases should be specified as comma separated string, with each comma delimiting an alias. If it is not specified, the prefix (see [a]) is used as the default alias for the plugin.
network_address	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 "<DTLS_prefix>.create_function".

Table 20.2 **Properties for NDDS_Transport_DTLS_Property_t**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[a]	Property Value Description
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
parent.address_bit_count	Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, the address_bit_count should be set to 8.
parent.properties_bitmap	A bitmap that defines various properties of the transport to the *RTI Data Distribution Service* core. Currently, the only property supported is whether or not the transport plugin will always loan a buffer when *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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. Note: If you use a value greater than the default, the *DomainParticipant's* **buffer_size** (in the RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7)) should also be changed.
parent.allow_interfaces	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.

Table 20.2 **Properties for NDDS_Transport_DTLS_Property_t**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[a]	Property Value Description
parent.deny_interfaces	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.
send_socket_buffer_size	Size in bytes of the send buffer of a socket used for sending.
recv_socket_buffer_size	Size in bytes of the receive buffer of a socket used for sending.
ignore_loopback_interface	Prevents the Transport Plugin from using the IP loopback interface.
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.
transport_priority_mask	Mask for use of transport priority field.
transport_priority_mapping_low transport_priority_mapping_high	Low and high values of output range to IPv4 TOS.
recv_decode_buffer_size	Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data.
port_offset	Port offset to allow coexistence with non-secure UDP transport.

Table 20.2 **Properties for NDDS_Transport_DTLS_Property_t**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[a]	Property Value Description
dtls_handshake_resend_interval	DTLS handshake retransmission interval in milliseconds
dtls_connection_liveliness_ interval	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.
tls.verify.ca_file	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. ***ca_file or ca_path must be specified; both may be specified.***
tls.verify.ca_path	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. ***ca_file or ca_path must be specified; both may be specified.***
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify.verify_peer	If non-zero, use mutual authentication when performing TLS handshake (default). If zero, only the reader side will present a certificate, which will be verified by the writer side.
tls.verify.verify_callback	This can be set to one of three values: -- "default" selects the default callback **NDDS_Transport_TLS_default_verify_callback()** -- "verbose" selects the verbose callback **NDDS_Transport_TLS_verbose_verify_callback()** -- "none" requests no callback be registered
tls.cipher.cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string.
tls.cipher.dh_param_files	List of available Diffie-Hellman (DH) key files. For example: "foo.h:512,bar.h:256" means: dh_param_files[0].file = foo.pem, dh_param_files[0].bits = 512, dh_param_files[1].file = bar.pem, dh_param_files[1].bits = 256

Table 20.2 **Properties for NDDS_Transport_DTLS_Property_t**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[a]	Property Value Description
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).

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

20.5 Explicitly Instantiating a WAN or Secure Transport Plugin

As described on page 20-1, 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 Section 20.2).

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 Section 20.2.
- 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 Section 20.5.1 and Section 20.5.2 for details.

To instantiate a WAN or Secure Transport prior to explicitly registering it with **NDDSTransportSupport::register_transport()**, use one of the following functions:

```
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 online (HTML) documentation for details on these functions.

20.5.1 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 9.1, “Header Files to Include for RTI Data Distribution Service (All Architectures),” on page 9-2).

  ```
  #include "ndds/ndds_transport_secure_wan.h"
  ```

  Assuming that *RTI Secure WAN Transport* is installed in the same directory as *RTI Data Distribution Service* (see Table 9.2, “Include Paths for Compilation (All Architectures),” on page 9-3), 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**:

  ```
  #include <openssl/ssl.h>
  #include <openssl/x509.h>  (if accessing certificate functions)
  etc.
  ```

  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>**.

20.5.2 Additional Libraries

To use the Secure WAN Transport API, you must link in additional libraries, which are listed in the Platform Notes (in the appropriate section for your architecture). Refer to Section 9.3.1 for differences between shared and static libraries.

20.5.3 Compiler Flags

No additional compiler flags are required.

Part 5: RTI Persistence Service

The material in this part of the manual describes *RTI Persistence Service,* which is included with *RTI Data Distribution Service Professional Edition* and *Elite Edition*. It saves 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.

- ❏ Chapter 21: Introduction to RTI Persistence Service
- ❏ Chapter 22: Configuring RTI Persistence Service
- ❏ Chapter 23: Running RTI Persistence Service

Chapter 21 Introduction to RTI Persistence Service

RTI Persistence Service is an *RTI Data Distribution Service* application that saves 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.

RTI 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, *RTI 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 Release Notes for the list of platforms and relational databases that have been tested.

When configured to run in TRANSIENT mode, *RTI 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 Chapter 2: Data-Centric Publish-Subscribe Communications. You should also have already read Chapter 11: Mechanisms for Achieving Information Durability and Persistence.

Chapter 22 Configuring RTI Persistence Service

To use *RTI Persistence Service*:

1. Modify your *RTI Data Distribution Service* applications.
   - The DURABILITY QosPolicy (Section 6.5.6) controls whether or not, and how, published samples are stored by *RTI Persistence Service* for delivery to late-joining *DataReaders*. See Data Durability (Section 11.5).
     - 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 DURABILITY SERVICE QosPolicy (Section 6.5.7), which can be used to configure *RTI Persistence Service*.

   By default, the History and ResourceLimits QosPolicies for a Persistence Service DataReader (PRSTDataReader) and Persistence Service DataWriter (PRST-DataWriter) with topic 'A' will be configured using the DurabilityService QosPolicy of the first-discovered *DataWriter* publishing 'A'. (For more information on the PRSTDataReader and PRSTDataWriter, see RTI Persistence Service (Section 11.5.1).) These values will overwrite the values specified in the XML file (unless you use the tag **<use_durability_service>** in the persistence group definition, see Section 22.7).

2. Create a configuration file or edit an existing file, as described in XML Configuration File (Section 22.2).

3. Start *RTI Persistence Service* with your configuration file, as described in Starting RTI Persistence Service (Section 23.1). 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).

22.1 How to Load the XML Configuration

RTI 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 *RTI Data Distribution Service* (see Chapter 15 in the *RTI Data Distribution Service User's Manual*).

- **$NDDSHOME/resource/qos_profiles_4.5x[1]/xml/NDDS_QOS_PROFILES.xml**

  This file contains the *RTI Data Distribution Service* default QoS values; it is loaded automatically if it exists. *(First to be loaded.)*

- **File specified in 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 *RTI Persistence Service*.

- **<*RTI Persistence Service executable location*>/../../resource/xml/RTI_PERSISTENCE_SERVICE.xml**

  This file contains the default *RTI 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.

1. *x* stands for the version letter of the current release.

- ❏ **File specified using the command line option, -cfgFile**

  The command-line option **-cfgFile** (see Table 23.1 in the Getting Started Guide) can be used to specify a configuration file.

22.2 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 chapter:

- ❏ QoS Configuration (Section 22.3)
- ❏ Configuring the RTI Persistence Service Application (Section 22.4)
- ❏ Configuring the RTI Persistence Service Application (Section 22.4)
- ❏ Configuring the Persistent Storage (Section 22.5)
- ❏ Configuring Participants (Section 22.6)
- ❏ Creating Persistence Groups (Section 22.7)

Example Configuration File

```
<?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">
            <writer_qos name="WriterQos1">
                <history>
                    <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
                </history>
            </writer_qos>
            <reader_qos name="ReaderQos1">
                <reliability>
                    <kind>DDS_RELIABLE_RELIABILITY_QOS</kind>
                </reliability>
                <history>
                    <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
```

```
                </history>
            </reader_qos>
        </qos_profile>
    </qos_library>

    <!-- PERSISTENCE SERVICE SECTION -->
    <persistence_service name="Srv1">

        <!-- PERSISTENT STORAGE SECTION -->
        <persistent_storage>
            <filesystem>
                <directory>/tmp</directory>
                <file_prefix>PS</file_prefix>
            </filesystem>
        </persistent_storage>
        <!-- DOMAIN PARTICIPANT 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>
                <writer_qos base_name="QosLib1::QosProfile1"/>
                <reader_qos base_name="QosLib1::QosProfile1"/>
            </persistence_group>

        </participant>
    </persistence_service>
</dds>
```

22.2.1 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 22.1.

Table 22.1 **Supported Tag Values**

Type	Format	Notes
DDS_Boolean	yes, 1, true, BOOLEAN_TRUE or DDS_BOOLEAN_TRUE: these all mean TRUE	Not case-sensitive
	no, 0, false, BOOLEAN_FALSE or DDS_BOOLEAN_FALSE: these all mean FALSE	
DDS_Enum	A string. Legal values are those listed in the online (HTML) documentation for the C or Java API.	Must be specified as a string. (Do not use numeric values.)
DDS_Long	-2147483648 to 2147483647 or 0x80000000 to 0x7fffffff or LENGTH_UNLIMITED or DDS_LENGTH_UNLIMITED	A 32-bit signed integer
DDS_UnsignedLong	0 to 4294967296 or 0 to 0xffffffff	A 32-bit unsigned integer
String	UTF-8 character string	All leading and trailing spaces are ignored between two tags

22.2.2 XML Validation

22.2.2.1 Validation at Run-Time

RTI Persistence Service validates the input XML files using a builtin Document Type Definition (DTD). You can find a copy of the builtin DTD in **$(NDDSHOME)/resource/rti-persistenceservice/schema/rti_persistence_service.dtd**. (This is only a copy of what the *RTI 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 builtin DTD by using the XML tag, <!DOCTYPE>. For example, the following indicates that *RTI Persistence Service* must use a different DTD file to perform validation:

```
<!DOCTYPE dds SYSTEM
 "/local/usr/rti/dds/modified_rtipersistenceservice.dtd">
```

If you do not specify the DOCTYPE tag in the XML file, the builtin DTD is used.

The DTD path can be absolute, or relative to the application's current working directory.

22.2.2.2 Validation During Editing

RTI 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 **$(NDDSHOME)/resource/rtipersistenceservice/schema/rti_persistence_service.dtd** and **$(NDDSHOME)/resource/rtipersistenceservice/schema/rti_persistence_service.xsd**, respectively.

To include a reference to the XSD document in your XML file, use the attribute xsi:noNamespaceSchemaLocation in the **<dds>** tag. For example (in the following, replace **<NDDSHOME>** with the *RTI Data Distribution Service* installation directory):

```
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation=
"<NDDSHOME>/resource/rtipersistenceservice/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 *RTI Data Distribution Service* installation directory):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dds SYSTEM
"<NDDSHOME>/resource/rtipersistenceservice/schema/rti_persistence_service.dtd">
<dds>
   ...
</dds>
```

22.3 QoS Configuration

Each persistence group and participant has a set of DDS QoSs. There are six tags:

- <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:

```
<datawriter_qos name="DerivedWriterQos" base_qos_name="Lib::BaseWriter-
Qos">
    <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 Chapter 15: Configuring QoS with XML

The persistence groups and participants can use QoS libraries and profiles to configure their QoS values. For example:

```
<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 GROUP SECTION -->
            <persistence_group name="PerGroup1" filter="*">
                <single_publisher>true</single_publisher>
                <single_subscriber>true</single_subscriber>
                <datawriter_qos base_name="QosLib1::QosProfile1"/>
           </persistence_group>
        </persistence_service>
    </dds >
```

For more information about QoS libraries and profiles see Chapter 15: Configuring QoS with XML.

22.4 Configuring the RTI Persistence Service Application

Each execution of the *RTI Persistence Service* application is configured using the content of a persistence service application tag (<persistence_service>). When you start *RTI Persistence Service* (described in Section 23.1), you must specify which <persistence_service> tag to use to configure the service.

For example:

```
<dds>
    <persistence_service name="Srv1">
    ...
    </persistence_service>
</dds>
```

If you do not specify a service name when you start *RTI 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 *RTI Persistence Service* executions.

Table 22.2 lists the tags you can specify for a persistence service. For default values, please see the online (HTML) documentation.

Table 22.2 **Persistence Service Application Tags**

Tag	Description	Number of Tags Allowed

<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

<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 Section 22.5.	0 or 1
<participant>	For each <participant> tag, *RTI Persistence Service* creates two *DomainParticipants* on the same domain ID: one to subscribe to changes and one to publish changes. The QoS values used to configure both participants are the same, except for **WireProtocol.domain_id**. If WireProtocol.domain_id is not -1 (the default value), RTI Persistence Service uses WireProtocol.domain_id for the first domain *DomainParticipant* and **WireProtocol.domain_id**+1 for the second *DomainParticipant*.	1 or more
<synchronization>	A DDS_Boolean (see Table 22.1) that indicates if redundant persistence service instances should synchronize their states with one another. When set to TRUE, messages 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 <synchronize> must be set to true in every instance involved in the synchronization. Note: This 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 persistence service instances. Default: 0	0 or 1

22.5 Configuring the Persistent Storage

The <persistent_storage> tag is used to persist samples into permanent storage. If the <persistence_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 22.3 lists the tags that you can specify in <persistent_storage>.

Table 22.3 **Persistent Storage tags**

Tag	Description	Number of Tags Allowed
<filesystem>	When this tag is present, the topic data will be persisted into files. *This tag is required if <external_database> is not specified.*	0 or 1
<external_database>	When this tag is present, the topic data will be persisted in a relational database. *This tag is required if <filesystem> is not specified.*	0 or 1
<restore>	This DDS_Boolean (see Table 22.1) 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	0 or 1

Table 22.5 and Table 22.4 list the tags that you can specify in <filesystem> and <external_database> .

Relational Database Limitations: The ODBC storage does not support BLOBs. The maximum size for a serialized sample is 65535 bytes in MySQL and 4194304 bytes in TimesTen.

Table 22.4 **Filesystem tags**

Tag	Description	Number of Tags Allowed
<directory>	Specifies the directory of the files in which topic data will be persisted. There will be one file per PRSTDataWriter/PRSTDataReader pair. Default: current working directory	0 or 1
<file_prefix>	A name prefix associated with all the files created by *RTI Persistence Service*. Default: PS	0 or 1
<synchronization>	Determines the level of synchronization with the physical disk. This tag can take three values: • FULL: Every sample is written into physical disk as *RTI Persistence Service* receives it. • NORMAL: 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

Table 22.5 **External Database Tags**

Tag	Description	Number of Tags Allowed
<dsn>	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 UNIX/Linux systems. *This tag is required.*	1
<username>	Username to connect to the database. Default: no username is used	0 or 1
<password>	Password to connect to the database. Default: no username is used	0 or 1
<odbc_library>	Specifies the ODBC driver to load. By default, *RTI Data Distribution Service* will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).	0 or 1

22.6 Configuring Participants

An XML <persistence_service> tag will contain a set of domain participants. The persistence service will persist topics published in the domainIDs associated with these participants. For example:

```
<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 domain participants on domains 71 and 72. After the domain participants are created, the persistence service will monitor the discovery traffic looking for topics to persist.

The <domain_id> tag can be specified alternatively as an attribute of <participant>. For example:

```
<persistence_service name="Srv1">
    <participant name="Part1" domain_id="71">
        ...
    </participant>
</persistence_service>
```

Table 22.6 further describes the participant tags.

Table 22.6 **Participant Tags**

Tag	Description	Number of Tags Allowed
<domain_id>	Domain ID associated with the Participant. The domain ID can be specified as an attribute of the participant tag. Default: 0	0 or 1
<participant_qos>	Participant QoS. Default: DDS defaults	0 or 1
<persistence_group>	A persistence group describes a set of topics whose data that must be persisted by the persistence service.	1 or more

22.7 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:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="H*">
       ...
    </persistence_group>
</participant>
```

In the previous example, the persistence group 'PerGroup1' is associated with all the topics published in 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 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 22.7 further describes the persistence group tags. For default values, please see the online (HTML) documentation.

Table 22.7 **Persistence Group Tags**

Tag	Description	Number of Tags Allowed
<filter>	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 <persistence_group> as well. Default: *	0 or 1
<content_filter>	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 online documentation (from the Modules page, select DDS API Reference, Queries and Filters Syntax. Default: no expression)	0 or 1
<propagate_dispose>	A DDS_Boolean (see Table 22.1) that controls whether or not the persistence service propagates dispose messages from DataWriters to *DataReaders*. Default: 1	0 or 1
<propagate_unregister>	A DDS_Boolean (see Table 22.1) that controls whether or not the persistence service propagates unregister messages from *DataWriters* to *DataReaders*. Default: 0	0 or 1
<single_publisher>	A DDS_Boolean (see Table 22.1) that indicates if the persistence service should create one Publisher per persistence group or one Publisher per PRSTDataWriter inside the persistence group. Default: 1	0 or 1
<single_subscriber>	A DDS_Boolean (see Table 22.1) that indicates if the persistence service should create one Subscriber per persistence group or one Subscriber per PRSTDataReader in the persistence group. Default: 1	0 or 1
<use_durability_ service>	A DDS_Boolean (see Table 22.1) 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. Default: 1	0 or 1

Table 22.7 **Persistence Group Tags**

Tag	Description	Number of Tags Allowed
<share_database_ connection>	A DDS_Boolean (see Table 22.1) 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 <single_publisher> is 0 and <share_database_connection> is 1, there is a single database connection per group. All the PRSTDataWriters will share the same connection. When <single_publisher> is 1 or <share_database_connection> 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 <external_database> in <persistent_storage>. Default: 0	0 or 1
<reader_checkpoint_ frequency>	This property controls how often (expressed as a number of samples) the PRSTDataReader state is stored in the database. The PRSTDataReaders are the DDS 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 samples. The persistence service will send these duplicates samples on the wire but they will be filtered by the DDS DataReaders and they will not be propagated to the application. This property is only applicable when the persistence service operates in persistent mode (the <persistent_storage> tag is present). Default: 1	0 or 1

Table 22.7 **Persistence Group Tags**

Tag	Description	Number of Tags Allowed
<writer_in_memory_state>	A DDS_Boolean (see Table 22.1) 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 <persistent_storage> 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 sample. This mode provides the best performance. However, the restore operation will be slower and the maximum number of 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 samples that a PRSTDataWriter can manage will not be limited by the available physical memory unless the underlying persistent storage is an in-memory database (TimesTen). Default: For KEEP_LAST or **ResourceLimitsQosPolicy.max_samples** != DDS_UNLIMITED_LENGTH, the default is 1. Otherwise, the default is 0.	0 or 1
<topic_qos>	Topic QoS. Default: DDS defaults	0 or 1
<publisher_qos>	Publisher QoS. Default: DDS defaults	0 or 1
<subscriber_qos>	Subscriber QoS Default: DDS defaults	0 or 1
<writer_qos>	PRSTDataWriter QoS[a] Default: DDS defaults	0 or 1
<reader_qos>	PRSTDataReader QoS[a] Default: DDS defaults	0 or 1

a. These fields cannot be set and are assigned automatically: protocol.virtual_guid, protocol.rtps_object_id, durability.kind.

22.7.1 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:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="*">
     ...
        <publisher_qos base_qos_name="QosLib1::PubQos1"/>
        <subscriber_qos base_qos_name="QosLib1::SubQos1"/>
        <writer_qos base_qos_name="QosLib1::WriterQos1"/>
        <reader_qos base_qos_name="QosLib1::ReaderQos1"/>
    ...
    </persistence_group>
</participant>
```

For instance, the number of samples saved by *RTI Persistence Service* is configurable through the HISTORY QosPolicy (Section 6.5.8) of the PRSTDataWriters.

If a QoS tag is not specified the persistence service will use the corresponding *RTI Data Distribution Service* default values (Section 22.7.2 describes an exception to this rule).

22.7.2 Durability Service Qos Policy

The DURABILITY SERVICE QosPolicy (Section 6.5.7) associated with a *DataWriter* is used to configure the HISTORY and the RESOURCE_LIMITS associated with the PRST-DataReaders and PRSTDataWriters.

By default, the HISTORY and RESOURCE_LIMITS of a PRSTDataReader and *PRST-DataWriter* with topic 'A' will be configured using the DURABILITY_SERVICE value of the first discovered *DataWriter* publishing 'A'. These values will overwrite the values specified in the XML file.

To not overwrite the XML values, you can use the tag <use_durability_service> in the persistence group definition:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="*">
    ...
        <use_durability_service/>0</ use_durability_service>
    ...
    </persistence_group>
</participant>
```

22.7.3 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 PRST-DataReader, use the tags <single_publisher> and <single_subscriber>, as follows:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="*">
    ...
        <single_publisher/>0</single_publisher>
        <single_subscriber/>0</single_subscriber>
    ...
    </persistence_group>
</participant>
```

22.7.4 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.

Chapter 23 Running RTI Persistence Service

This chapter describes how to start and stop *RTI Persistence Service*.

You can run *RTI 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.

23.1 Starting RTI Persistence Service

The script to run *RTI Persistence Service*'s executable is located in **$NDDSHOME/scripts**.

```
RTI Persistence Service v4.5c
Usage: rtipersistenceservice [options]
Options:
 -cfgFile        <file>  Configuration file. This parameter is optional
                         since the configuration can be loaded from
                         other locations
 -cfgName        <name>  Configuration name. This parameter is required
                         and it is used to find a <persistence_service>
                         matching tag in the configuration files
 -appName        <name>  Application name
                         Used to name the DDS domain participants
                         Default: -cfgName
 -domainId       <int>   DDS domain ID for the domain participants
                         created by the service
                         Default: Use XML value
 -restore        <0|1>   Indicates whether or not persistence service
                         must restore its state from the persistent
                         storage
                         Default: Use XML value
```

```
-verbosity      [0-6]   RTI Persistence Service verbosity
                         * 0 - silent
                         * 1 - exceptions (DDS and service)
                         * 2 - warnings (service)
                         * 3 - information (service)
                         * 4 - warnings (DDS and service)
                         * 5 - tracing (service)
                         * 6 - tracing (DDS and service)
                        Default: 1 (exceptions)
-version                Prints the RTI Persistence Service version
-licenseFile    <file>  License file. This parameter is optional
-help                   Displays this information
```

Note: -cfgName is required.

The command-line options are described in Table 23.1.

23.2 Stopping RTI Persistence Service

To stop RTI Persistence Service: press **Ctrl-C**. *RTI Persistence Service* will close all files and perform a clean shutdown.

Table 23.1 **Command-Line Options**

Command-line Option	Description
-appName <string>	Assigns an application name to the *DomainParticipants* created by *RTI Persistence Service*. The participant name format is as follows: **RTI Persistence Service: <appName>: <participant-Name>(<pub|sub>)** Default: **-cfgName value**
-cfgFile <string>	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 Section 22.1 for further details.

Table 23.1 **Command-Line Options**

Command-line Option	Description
`-cfgName <string>`	**Required.** Selects a persistence service configuration. The same configuration files can be used to configure multiple persistence services. Each persistence service will load its configuration from a different <persistence_service> tag based on the name specified with this option. f it is not specified, *RTI Persistence Service* will print the list of available configurations and then exit.
`-domainId`	Sets the domain ID for the *DomainParticipants* created by *RTI Persistence Service*. If this option is not specified, the value in the <participant> XML tag (see Table 22.6 on page 22-12) is used.
`-help`	Prints the *RTI Persistence Service* version and list of command-line options.
`-licenseFile <file>`	Specifies the license file (path and filename). Only applicable to licensed versions of *RTI Persistence Service*. If not specified, *RTI Persistence Service* looks for the license as described in the *RTI Persistence Service Installation Guide*.
`-restore < 0|1>`	Indicates whether or not *RTI Persistence Service* must restore its state from the persistent storage. 0 = do not restore; 1 = do restore. If this option is not specified, the corresponding XML value in the <persistent_storage> tag (see Table 22.3 on page 22-10) is used.
`-verbosity`	*RTI Persistence Service* verbosity: 0 - No verbosity 1 - Exceptions (DDS and *RTI Persistence Service*) (default) 2 - Warning (*RTI Persistence Service*) 3 - Information (*RTI Persistence Service*) 4 - Warning (DDS and *RTI Persistence Service*) 5 - Tracing (*RTI Persistence Service*) 6 - Tracing (DDS and *RTI Persistence Service*) Each verbosity level, *n*, includes all the verbosity levels smaller than *n*.
`-version`	Prints the *RTI Persistence Service* version.

Part 6: RTI CORBA Compatibility Kit

The material in this part of the manual is only relevant if you have purchased *RTI CORBA Compatibility Kit*, an optional package that allows *RTI Data Distribution Service*'s code generator, *rtiddsgen*, to output type-specific code that is compatible with OCI's distribution of TAO and the JacORB distribution.

- Chapter 24: Introduction to RTI CORBA Compatibility Kit
- Chapter 25: Generating CORBA-Compatible Code with rtiddsgen
- Chapter 26: Supported IDL Types

Chapter 24 Introduction to RTI CORBA Compatibility Kit

RTI CORBA Compatibility Kit is an optional package that allows *RTI Data Distribution Service*'s code generator, *rtiddsgen*, to output type-specific code that is compatible with OCI's or DOC's distribution of TAO and the JacORB distribution.

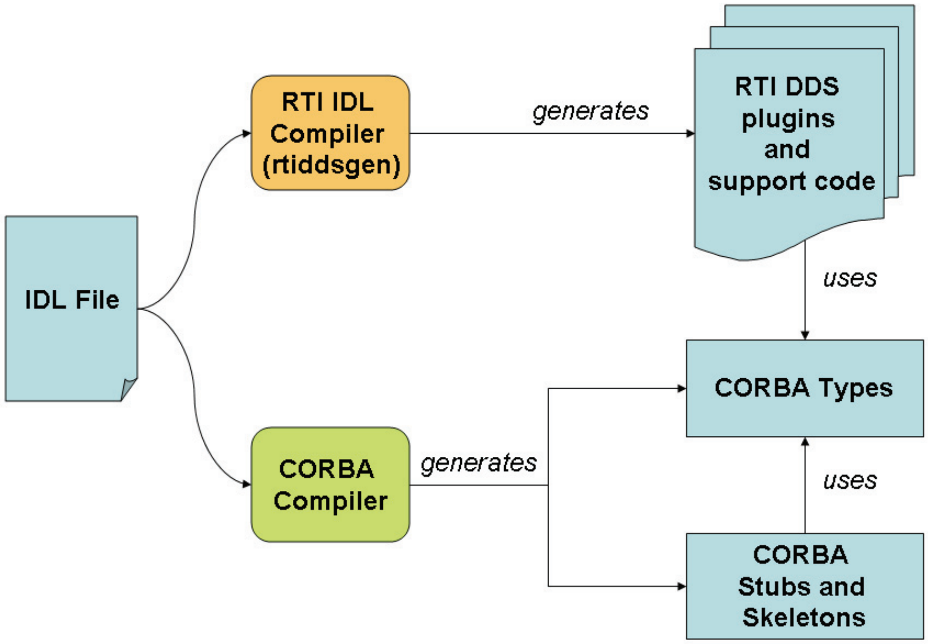
By having compatible data types, your applications can use CORBA *and RTI Data Distribution Service* APIs, with no type conversions required.

For more information about OCI's or DOC's distribution of TAO and JacORB, please refer to the documentation included with those distributions. Additional information can be found on OCI's TAO website (www.theaceorb.com), DOC's TAO website (http://www.dre.vanderbilt.edu), and JacORB's website (www.jacorb.org). TAO and JacOrb distributions that are compatible with this version of *RTI Data Distribution Service* are available from the RTI Customer Portal, accessible from www.rti.com/support.

In addition to this document, a simple example is available.

- ❑ **C++ using TAO:**
  - See the example in <RTI_Data_Distribution_Service_INSTALL_ROOT>/example/CPP/corba. Please read Instructions.pdf.
- ❑ **Java using JacORB:**
  - See the example in <RTI_Data_Distribution_Service_INSTALL_ROOT>/example/JAVA/corba. Please read Instructions.pdf.

The following figure shows the process of using common IDL files and types for both CORBA and DDS.



RTI CORBA Compatibility Kit is designed to be installed on top of *RTI Data Distribution Service*; this kit replaces the default version of *RTI Data Distribution Service*'s code generation tool, *rtiddsgen*. The replacement *rtiddsgen* includes support for the command-line option, **-corba**.

On the wire, the serialized version of the code for types generated using the **-corba** option is identical to the serialized version of the code for types generated without the option. As result, endpoints (*DataReaders* or *DataWriters*) using type support code generated with **-corba** can fully communicate with endpoints using type support code generated without **-corba**.

Chapter 25 Generating CORBA-Compatible Code with rtiddsgen

The *RTI CORBA Compatibility Kit* enables *RTI Data Distribution Service*'s IDL compiler, *rtiddsgen*, to produce type-specific code that is compatible with OCI's distribution of TAO for C++ and with JacORB for Java.

When using *rtiddsgen*, specify the **-corba** option on the command line to generate compatible code. The **-corba** option enables the use of data structures for both CORBA and DDS API calls without requiring any translation: the IDL-to-language mapping is the same for both.

There are some trade-off's to consider:

- ❏ While the **-corba** option provides the benefit of CORBA-compatible type-specific code, it does not provide support for bitfields, pointers and ValueTypes.
- ❏ For complex types such as sequences and strings, the memory management is different when the **-corba** option is used. When code is generated without the option, the memory needed for the type is pre-allocated at system initialization. When code is generated with the option, the memory is allocated when it is needed, so memory allocation system calls may occur while the system is in steady state.
- ❏ Without the **-corba** option, access to data fields within types may be faster under some circumstances. CORBA-compatible types require the use of accessor methods. When **-corba** is *not* used, while the accessor methods are provided for convenience but they can be bypassed and the data can be accessed directly. This direct access is available to the user as well as to the DDS middleware internal implementation code. As a result, depending on the complexity of the types used, overall system latency could be lower when using non-compatible types (that is, when **-corba** is not used).

The following sections describe how to use the *RTI CORBA Compatibility Kit*. In addition to these instructions, a simple example is available.

- **C++ using TAO:**
  - Generating C++ Code (Section 25.1)
  - See the example in <RTI_Data_Distribution_Service_INSTALL_ROOT>/example/CPP/corba. Please read Instructions.pdf.
- **Java using JacORB:**
  - Generating Java Code (Section 25.2)
  - See the example in <RTI_Data_Distribution_Service_INSTALL_ROOT>/example/JAVA/corba. Please read Instructions.pdf.

25.1 Generating C++ Code

To generate CORBA-compatible type-specific code, first run TAO's code generator, **tao_idl**, on the IDL file containing your data types. If you followed the TAO distribution compilation instructions contained in this document, the tao_idl compiler executable will be in the TAO install directory under **<ACE_ROOT>/bin**.

```
<ACE_ROOT>/bin/tao_idl <IDL file name>.idl
```

This will generate CORBA support files for your data types. The generated file will have a name matching the pattern **<IDL file name>C.h** and will contain the type definitions. Pass this header file as a parameter to *rtiddsgen* to generate the DDS support code for the data types.

```
rtiddsgen -language C++ -corba <IDL file name>C.h -example \
<architecture> <IDL file name>.idl
```

The optional **-example <architecture>** flag will generate code for a DDS publisher and a DDS subscriber. It will also generate an **.mpc** file (and an **.mwc** file for Windows) that can be used with TAO's Makefile, Project and Workspace Creator (MPC) to generate a makefile or a Visual Studio project file for your DDS-CORBA application. The **.mpc** file is meant to work out-of-the-box with the DDS-CORBA C++ Message example only, so you will have to modify it to compile your custom application. Please refer to the DDS-CORBA C++ example for more information about using MPC (see Instructions.pdf).

25.2 Generating Java Code

To generate Java CORBA-compatible type specific code, first run the JacORB code generator on the IDL file containing your data types.

```
<JacORB install dir>/bin/idl <IDL file name>.idl
```

After generating the CORBA code for the IDL types run *rtiddsgen* as follows:

```
rtiddsgen -language Java -corba -example <architecture> \
<IDL file name>.idl
```

The optional **-example <architecture>** flag will generate code for a DDS publisher and a DDS subscriber. It will also generate a makefile specific to your architecture that can be used to compile the example using the publisher and subscriber code generated.

To form a complete code set, use the type class generated by the CORBA IDL compiler and the files generated by *rtiddsgen*.

Chapter 26 Supported IDL Types

Table 26.1 lists the IDL types supported when using the **–corba** option.

Table 26.1 Supported IDL Types when Using rtiddsgen -corba

IDL Construct	Support
Modules	Supported
Interfaces	Ignored
Constants	Supported
Basic Data Types	Supported
Enums	Supported
String Types	Supported
Wide String Types	Supported
Struct Types	Supported Note: In-line nested structures are not supported (whether using -corba or not). See Note 1.
Fixed Types	Ignored
Union Types	Supported
Sequence Types	Supported Note: Sequences of anonymous sequences are not supported. See Note 2.
Array Types	Supported
Typedefs	Supported
Any	Not Supported. Note that *rtiddsgen* does not ignore them. This construct cannot be in the IDL file.
Value Types	Ignored

Table 26.1 Supported IDL Types when Using rtiddsgen -corba

IDL Construct	Support
Exception Types	Ignored
Type Code	Supported. With DDS, DDS type codes should be used. With CORBA, CORBA type codes should be used.

- Note 1

  Inline nested structures, such as the following example, are *not* supported.

```
struct Outer {
    short outer_short;

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

- Note 2

  Sequences of anonymous Sequences are *not* supported. This kind of type will be banned in future revisions of CORBA. For example, the following is *not* supported:

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

  Instead, sequences of sequences can be supported using typedef definitions. For example, this *is* supported:

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

Part 7: RTI RTSJ Extension Kit

The material in this part of the manual is only relevant if you have purchased *RTI RTSJ Extension Kit*, an optional package that allows you to configure *RTI Data Distribution Service* applications to use Real-Time Specification for Java (RTSJ)-specific thread types and memory areas.

- ❏ Chapter 27: Introduction to RTI RTSJ Extension Kit
- ❏ Chapter 28: Using RTI RTSJ Extension Kit

Chapter 27 Introduction to RTI RTSJ Extension Kit

The *RTI RTSJ Extension Kit* is an optional package that allows you to configure *RTI Data Distribution Service* applications to use Real-Time Specification for Java (RTSJ)-specific thread types and memory areas. In particular, the threads used by *RTI Data Distribution Service* can be configured so that they are never interrupted by a Java virtual machine's garbage collector—greatly improving the application's determinism.

For more information on RTSJ, please refer to the documentation available on the official web site: www.rtsj.org.

Additional documentation for the *RTI RTSJ Extension Kit* is available in the online documentation: <RTI_Data_Distribution_Service_INSTALL_ROOT>/ReadMe.html.

A simple example is also available:
<RTI_Data_Distribution_Service_INSTALL_ROOT>/example/JAVA/rtsj. Please read Instructions.pdf.

Chapter 28 Using RTI RTSJ Extension Kit

The kit includes a JAR file and associated electronic documentation, including API documentation in HTML and PDF formats and example code. The JAR file is provided in two versions, release and debug, named **nddsrtsj.jar** and **nddsrtsjd.jar**, respectively. These must be used in addition to the libraries provided with *RTI Data Distribution Service* itself. If you are using the *RTI Data Distribution Service release* JAR, we recommend that you also use the RTSJ *release* JAR, and likewise for the debug JAR, although this is not a requirement.

Detailed API documentation is available in HTML format, accessible here:
<RTI_Data_Distribution_Service_INSTALL_ROOT>/ReadMe.html.

A simple example is also available here:
<RTI_Data_Distribution_Service_INSTALL_ROOT>/example/JAVA/rtsj.
Please read Instructions.pdf.

Part 8: RTI TCP Transport

RTI TCP Transport is only available on specific architectures. See the Platform Notes for details.

Out of the box, *RTI Data Distribution Service* 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 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.

RTI TCP Transport enables participant discovery and data exchange using the TCP protocol (either on a local LAN, or over the public WAN). *RTI TCP Transport* allows *RTI Data Distribution Service* 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, an *RTI Data Distribution Service* application uses multiple UDP ports for communication, which may make it unsuitable for deployment across firewalled networks).

This part of the User’s Manual contains the following chapter:

❏ Chapter 29: Configuring the RTI TCP Transport

Chapter 29 Configuring the RTI TCP Transport

This chapter explains how to use and configure the *RTI TCP Transport*:

- ❏ TCP Communication Scenarios (Section 29.1)
- ❏ Configuring the RTI TCP Transport (Section 29.2)

29.1 TCP Communication Scenarios

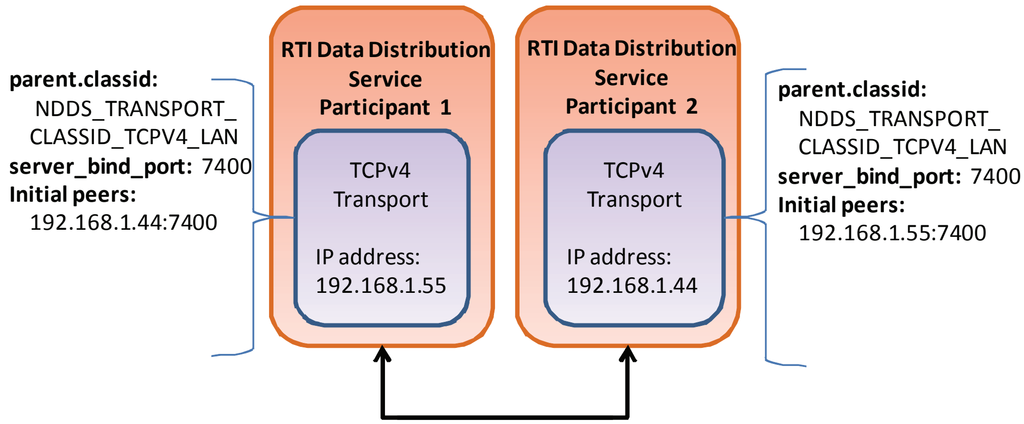
RTI TCP Transport can be used to address multiple communication scenarios that go from simple communication within a single LAN to complex communication scenarios across LANs where NATs and firewalls may be involved. This section describes these scenarios:

- ❏ Communication Within a Single LAN (Section 29.1.1)
- ❏ Symmetric Communication Across NATs (Section 29.1.2)
- ❏ Asymmetric Communication Across NATs (Section 29.1.3)

29.1.1 Communication Within a Single LAN

RTI TCP Transport can be used as an alternative to UDPv4 to communicate DDS applications running inside the same LAN. Figure 29.1 shows how to configure the TCP transport in this scenario.

Figure 29.1 **Communication within a Single LAN**



parent.classid and server_bind_port 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 29.1 on page 29-11.

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 Section 29.2.1.

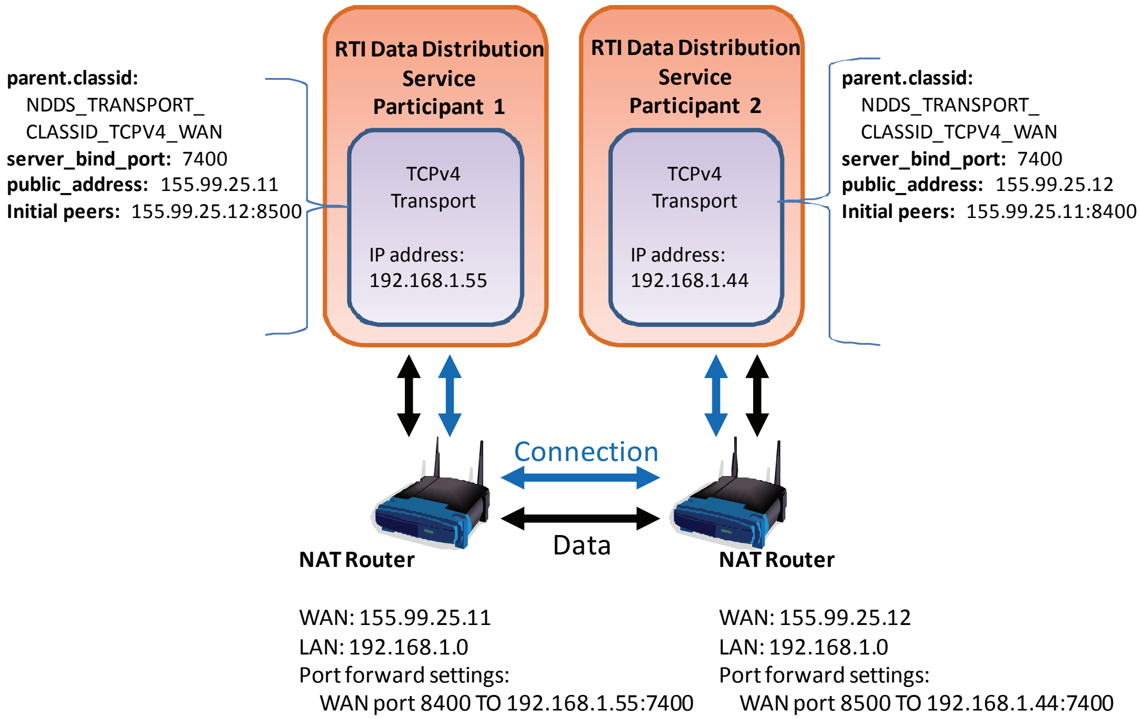
Note: Unlike the UDPv4 transport, you must specify the initial peers, because multicast cannot be used with TCP.

29.1.2 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 *RTI Data Distribution Service* application can initiate TCP connections with other applications. Figure 29.2 shows how to configure the TCP transport in this scenario.

Figure 29.2 **Symmetric Communication Across NATs**



Notice that initial peers refer to the public address of the remote LAN where the *RTI Data Distribution Service* application is deployed and not the private address of the node where the application is running. In addition, the transport associated with an *RTI Data Distribution Service* instance will have to be configured with its public address (public_address) so that this information can be propagated as part of the discovery process.

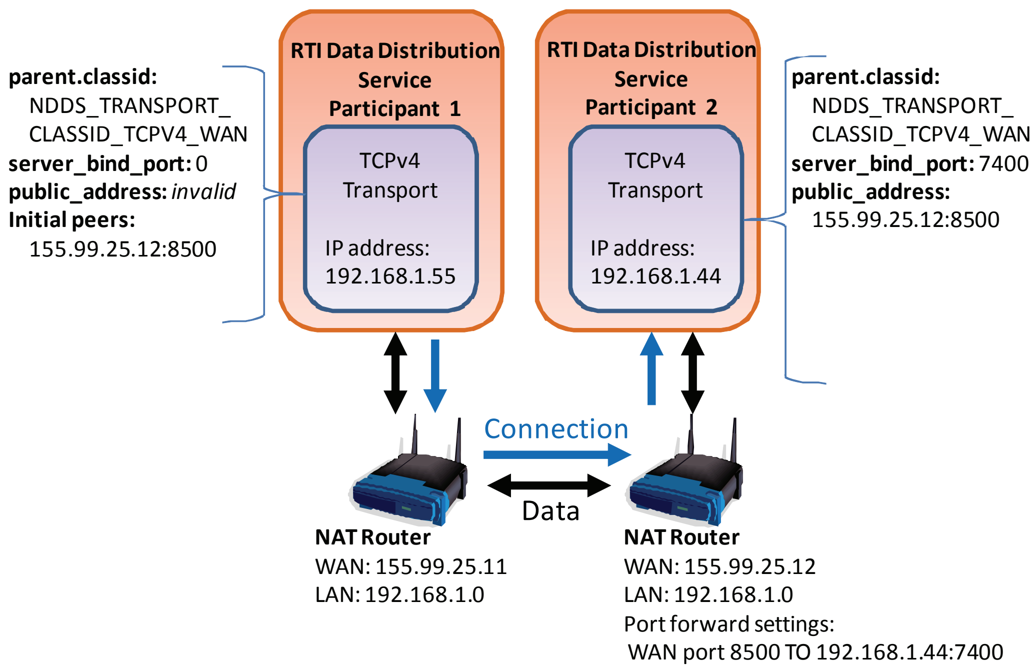
Because the public address and port of the *RTI Data Distribution Service* instances must be known before the communication is established, the NAT Routers will have to be configured statically to translate (forward) the private server_bind_port into a public **port**. This process is known as "static NAT" or "port forwarding" and 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.

29.1.3 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 *RTI Data Distribution Service* 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 29.3 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 29.3 **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 *RTI Data Distribution Service* 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. For this configuration, you should consider using *RTI Routing Service* (available for purchase as a separate product).

- 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.

29.2 Configuring the RTI TCP Transport

RTI TCP Transport is distributed as a both shared and static library in ***<RTI Data Distribution Service installation directory>*/lib/*<architecture>*.** The library is called **nddstransporttcp**.

Mechanisms for Configuring the Transport:

- ❏ **By explicitly instantiating a new transport** (see Section 29.2.2) and then registering it with the *DomainParticipant* (see Section 13.7). (Not available in the Java and .NET APIs.)
- ❏ **Through the Property QoS policy** of the *DomainParticipant* (on UNIX, Solaris and Windows systems only). This process is described in Section 29.2.3.

This section describes:

- ❏ Choosing a Transport Mode (Section 29.2.1)
- ❏ Explicitly Instantiating the TCP Transport Plugin (Section 29.2.2)
- ❏ Configuring the RTI TCP Transport with the Property QosPolicy (Section 29.2.3)
- ❏ Setting the Initial Peers (Section 29.2.4)
- ❏ TCP/TLS Transport Properties (Section 29.2.5)

29.2.1 Choosing a Transport Mode

When you configure the TCP transport, you must choose one of the following types of communication:

- ❏ **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 Section 19.3.

- ❑ **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.

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 Section 29.2.1) or in the parent.classid field of the Properties QoS (see Section 29.2.3). Your choice of transport mode will also be reflected in the prefix you use for setting the initial peers (see Section 29.2.4).

29.2.2 Explicitly Instantiating the TCP Transport Plugin

As described on page 29-6, 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 Section 29.2.3).

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 Section 29.2.3.
- ❑ 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 Section 29.2.2.1 and Section 29.2.2.2 for details.

To instantiate a TCP transport:

1. Include the extra header file described in Section 29.2.2.1.
2. Instantiate a new transport by calling **NDDS_Transport_TCPv4_new()**:

```
NDDS_Transport_Plugin* NDDS_Transport_TCPv4_new (
    const struct NDDS_Transport_TCPv4_Property_t * property_in)
```

3. Register the transport by calling **NDDSTransportSupport::register_transport()**.

See the online (HTML) documentation for details on these functions and the contents of the NDDS_Transport_TCPv4_Property_t structure.

29.2.2.1 Additional Header Files and Include Directories

To use the *RTI TCP Transport* API, you must include an extra header file (in addition to those in Table 9.1, “Header Files to Include for RTI Data Distribution Service (All Architectures),” on page 9-2):

```
#include "ndds/transport_tcp/transport_tcp_tcpv4.h"
```

Since *RTI TCP Transport* is in the same directory as *RTI Data Distribution Service* (see Table 9.2, “Include Paths for Compilation (All Architectures),” on page 9-3), no additional include paths need to be added for the *RTI TCP Transport* API. If this is not the case, you will need to specify the appropriate include path.

29.2.2.2 Additional Libraries and Compiler Flags

To use the *RTI TCP Transport*, you must add the **nddstransporttcp** library to the link phase of your application. There are four different kind of libraries, depending on if you want a debug or release version, and static or dynamic linking with *RTI Data Distribution Service*.

For UNIX- based 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-based systems, the libraries are:

NDDSTRANSPORTTCP.LIB — Release version, dynamic libraries

NDDSTRANSPORTTCPD.LIB — Debug version, dynamic libraries

NDDSTRANSPORTTCPZ.LIB — Release version, static libraries

NDDSTRANSPORTTCPZD.LIB — Debug version, static libraries

Notes for using TLS:

To use either TLS mode (see Section 29.2.1), 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 (LD_LIBRARY_PATH environment variable on UNIX/Solaris systems, or Path environment variable on Windows systems).

If you already have **$NDDSHOME/lib/<*architecture*>** in your library search path, no extra steps are needed to use TLS once *RTI TLS Support* is installed.

Even if you link everything statically, you must make sure that the location for **$NDDSHOME/lib/<*architecture*>** (or wherever the TLS library is located) is in your search path. The TLS library is loaded dynamically, even if you use static linking for everything else.

Your search path must also include the location for the Openssl library, which is used by the TLS library.

29.2.3 Configuring the RTI TCP Transport with the Property QosPolicy

The PROPERTY QosPolicy (DDS Extension) (Section 6.5.15) 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 Section 4.1.7. 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 Chapter 15: Configuring QoS with XML. This causes *RTI Data Distribution Service* to dynamically load the TCP transport library at run time and then implicitly create and register the transport plugin.

To add name/value pairs to the Property QoS policy, use the **add_property()** operation:

```
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 6.48, “PropertyQoSPolicyHelper Operations,” on page 6-139, as well as the online (HTML) documentation.

The ‘name’ part of the name/value pairs is a predefined string. The property names for the *RTI TCP Transport* are described in Table 29.1, “Properties for NDDS_Transport_TCPv4_Property_t,” on page 29-11.

Here are the basic steps, taken from the example Hello World application (for details, please see the example application.)

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.TCPv4.tcp1",
        DDS_BOOLEAN_FALSE);
```

   b. Specify the transport plugin library.

```
DDSPropertyQosPolicyHelper::add_property (participant_qos.property,
        "dds.transport.TCPv4.tcp1.library", "nddstransporttcp",
        DDS_BOOLEAN_FALSE);
```

   c. Specify the transport's 'create' function.

```
DDSPropertyQosPolicyHelper::add_property (participant_qos.property,
        "dds.transport.TCPv4.tcp1.create_function",
        "NDDS_Transport_TCPv4_create",
        DDS_BOOLEAN_FALSE);
```

   d. Set the transport to work in a WAN configuration with a public address:

```
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.public_address", "182.181.2.31",
        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);
```

Important! 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.

29.2.4 Setting the Initial Peers

Note: You must specify the initial peers (you cannot use the defaults because multicast cannot be used with TCP).

For *RTI 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>`

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:

```
setenv NDDS_DISCOVERY_PEERS tcpv4_wan://10.10.1.165:7400,
       tcpv4_wan://10.10.1.111:7400,tcpv4_lan://192.168.1.1:7500
```

When the TCP transport is configured for LAN communication (with the parent.classid 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 property).

When the TCP transport is configured for WAN communication (with the parent.classid 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.

29.2.5 TCP/TLS Transport Properties

Table 29.1 on page 29-11 describes the TCP and TLS transport properties.

Note: To use TLS, you also need *RTI TLS Support*, which is a separate package.

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
dds.transport.load_plugins *(Note: this does not take a prefix)*	**Required** Comma-separated strings indicating the prefix names of all plugins that will be loaded by *RTI Data Distribution Service*. For example: "**dds.transport.TCPv4.tcp1**". You will use this string as the prefix to the property names. See Footnote [1]. Note: you can load up to 8 plugins.
library	**Required** **Must be "nddstransporttcp".** This library needs to be in the path during run time for use by *RTI Data Distribution Service* (in the LD_LIBRARY_PATH environment variable on UNIX systems, in PATH for Windows systems).

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
create_function	Required **Must be "NDDS_Transport_TCPv4_create".**
aliases	Used to register the transport plugin returned by **NDDS_Transport_TCPv4_create()** (as specified by **<TCP_prefix>.create_function**) to the *DomainParticipant*. Aliases should be specified as a comma-separated string, with each comma delimiting an alias. Default: the transport prefix (see Footnote [1])
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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* 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 *RTI Data Distribution Service* from trying to gather too many buffers into a send call for the transport plugin. *RTI Data Distribution Service* requires all transport-plugin implementations to support a gather-send of least a minimum number of buffers. This minimum number is defined as NDDS_TRANSPORT_PROPERTY_GATHER_SEND_BUFFER_COUNT_MIN. Default: 128

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plugin. If you set this higher than the default, the *DomainParticipant's* **buffer_size** (in the RECEIVER_POOL QosPolicy (DDS Extension) (Section 8.5.7)) should also be changed. Default: 9216
parent.allow_interfaces_list	A list of strings, each identifying a range of interface addresses that can be used by the transport. Interfaces must be specified as comma-separated strings, with each comma delimiting an interface. For example: 10.10.*, 10.15.* If the list is non-empty, this "white" list is applied before parent.deny_interfaces_list. Default: All available interfaces are used.
parent.deny_interfaces_list	A list of strings, each identifying a range of interface addresses that will not be used by the transport. 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: `10.10.*` This "black" list is applied after parent.allow_interfaces_list and filters out the interfaces that should not be used. **Default:** No interfaces are denied
send_socket_buffer_size	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 or -1. The maximum value is operating system-dependent. Default: -1 (**setsockopt()** (or equivalent) will not be called to size the send buffer of the socket)

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
recv_socket_buffer_size	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 or -1. The maximum value is operating-system dependent. Default: -1 (**setsockopt()** (or equivalent) will not be called to size the receive buffer of the socket)
ignore_loopback_interface	Prevents the transport plugin from using the IP loopback interface. This property is ignored when parent.classid is NDDS_TRANSPORT_CLASSID_TCPV4_WAN or NDDS_TRANSPORT_CLASSID_TLSV4_WAN. Two values are allowed: 0: 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. 1: 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

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
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 no link is 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. Default: 1
transport_priority_mask	Mask for the transport priority field. This is used in conjunction with transport_priority_ mapping_low/transport_priority_ mapping_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. 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 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. Default transport_priority_mapping_low: 0 Default transport_priority_mapping_high: 0xFF
transport_priority_ mapping_high	

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
server_socket_backlog	The backlog parameter determines what is the maximum length of the queue of pending connections. Default: 5
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 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 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.
server_bind_port	Private IP port (inside the LAN) used by the transport to accept TCP connections. If this property is set to zero, the transport 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 WAN communication, this port must be forwarded to a public port in the NAT-enabled router that connects to the outer network. Default: 7400

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
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**)
write_buffer_allocation	Allocation settings applied to buffers used for asynchronous (non-blocking) write. 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** = 1000 ❑ **write_buffer_allocation.incremental_count** = 10 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, the low-level send command of the transport will fail; at that point *RTI Data Distribution Service* will take the appropriate action (retry to send or drop it), according to the application's QoS (if the transport is used for reliable communication, the data will still be sent eventually).
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**)

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
control_message_allocation	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: ❑ **control_message_allocation.initial_count** = 2 ❑ **control_message_allocation.max_count** = -1 (unlimited) ❑ **control_message_allocation.incremental_count** = -1 (number of messages will keep doubling on each allocation until it reaches **max_count**)
control_attribute_allocation	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: ❑ **control_attribute_allocation.initial_count** = 2 ❑ **control_attribute_allocation.max_count** = -1 (unlimited) ❑ **control_attribute_allocation.incremental_count** = -1 (number of attributes will keep doubling on each allocation until it reaches **max_count**)
force_asynchronous_send	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 and enqueue it in an internal send buffer. Data will be sent as soon as the low-level socket buffer has space. Normally setting it to 1 delivers better throughput in a fast network, but will result in a longer time to recover from various TCP error conditions. Setting it to 0 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, it 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). Default: 0

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
max_packet_size	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)
enable_keep_alive	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. On Solaris systems, most of the TCP keep-alive parameters can be changed though the kernel properties. Default: 0
keep_alive_time	Specifies the interval of inactivity in seconds that causes TCP to generate a KEEP_ALIVE message. This parameter is only supported on Linux architectures. Default: -1 (OS default value)
keep_alive_interval	Specifies the interval in seconds between KEEP_ALIVE retries. This parameter is only supported on Linux architectures. Default: -1 (OS default value)
keep_alive_retry_count	The maximum number of KEEP_ALIVE retries before dropping the connection. This parameter is only supported on Linux architectures. Default: -1 (OS default value)
disable_nagle	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

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
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: errors ❑ 0x02: warnings ❑ 0x04: local ❑ 0x08: remote ❑ 0x10: period ❑ 0x80: other (used for control protocol tracing) Default: -1 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. The default TCP transport verbosity is errors and warnings. Note: The option of 0x80 (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 / LIBNDDSTRANSPORTD.LIB**).
outstanding_connection_ cookies	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

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
outstanding_connection_ cookies_life_span	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).
tls.verify. ca_file	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. **To enable TLS, ca_file or ca_path is required;** both may be specified (at least one is required).
tls.verify. ca_path	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. **To enable TLS, ca_file or ca_path is required**; both may be specified (at least one is required).
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify. crl_file	Name of the file containing the Certificate Revocation List. File should be in PEM format.

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
tls.identity. certificate_chain	String containing an identifying certificate (in PEM format) or certificate chain (appending intermediate CA certs in order). **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.
tls.identity. certificate_chain_file	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** 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.
tls.identity. private_key_password	A string that specifies the password for private key.
tls.identity. private_key	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.
tls.identity. private_key_file	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.
tls.identity. rsa_private_key	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. At most one of **rsa_private_key** and **rsa_private_key_file** may be specified.
tls.identity. rsa_private_key_file	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. At most one of **rsa_private_key** and **rsa_private_key_file** may be specified.
tls.cipher. cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string.

Table 29.1 **Properties for NDDS_Transport_TCPv4_Property_t**

Property Name (prefix with 'dds.transport.TCPv4.tcp1.')[1]	**Description**
tls.cipher. dh_param_files	List of available Diffie-Hellman (DH) key files.
tls.cipher. engine_id	ID of OpenSSL cipher engine to request.

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.

Index

Symbols

@copy rtiddsgen directive 3-78
@key rtiddsgen directive 3-77
@resolve-name rtiddsgen directive 3-79
@top-level rtiddsgen directive 3-83
$NDDSHOME 3-110

A

absolute_generation_rank (DDS_SampleInfo) 7-67, 7-70
accept_unknown_peers 8-42, 8-45, 12-12
access control 14-14
access_scope (Presentation QoS) 6-80, 6-82
ACKNACK messages 6-18, 6-44, 10-3, 10-5, 10-27
acknowledgments 6-18, 6-44
ALIVE instance state 7-69
allocation of memory 7-78
  DataReaders 7-84
API message category 18-5
app ID 8-70
architectures supported i-xxii
as_Entity() 6-18, 7-23
assert_liveliness() 6-60, 6-127, 8-32
asynchronous data 7-3
autodispose_unregistered_instances (WriterDataLifeCycle QoS) 6-162
automatic participant ID 8-69
autoregister_instances (DataWriterResouceLimits QoS) 6-105

B

batching small samples 6-61, 6-86, 10-21
begin_coherent_changes() 6-44, 6-81
BEST_EFFORT (Reliable QoS) 6-144
best-effort delivery 10-2
blocking threads 6-105
blocking time 10-10
building applications 9-1, 9-3, 9-5
builtin data types
  definition 3-7
  See also data types.
  See also user data types.
built-in Subscriber 14-2, 14-12
  QoS 14-10
built-in topics 14-1
built-in transports 8-66
builtin_discovery_plugins (DiscoveryConfig QoS) 8-49
BuiltinTopicReaderResourceLimits 8-50
bundling messages 10-5

C

categories of messages 18-5
channel guard filter 16-3
cleanup_period 8-39
clear() 6-16, 7-19
client-server model 1-4
clock selection 8-75
coalescing samples 6-89
coherent sets 6-44
coherent_access (Presentation QoS) 6-80 *to* 6-81
command-line options for rtiddsgen 3-111
COMMUNICATION message category 18-5
communications models 1-3
  client-server 1-4
  data-centric 2-2
  DCPS 2-2, 2-11
  object-centric 2-2
  point-to-point 1-3
  publish-subscribe 1-4, 2-8
compatibility between QoS 6-51, 7-47
compiling applications 9-1
concurrent threads 6-105
Conditions 4-37
  example code 4-44
conserving CPU and bandwidth 10-24
contains_entity() 8-33
content filter 5-34
  syntax for 5-23
ContentFilteredTopics 6-105
  compile function 5-35
  creating 5-18
  custom filters 5-23, 5-34
  deleting 5-20
  evaluate function 5-37
  filter syntax 5-23
  finalize function 5-37
  introduction 5-15
  setting filter expressions 5-21
controlling queue depth 10-18
copy_from_topic_qos() 6-16, 6-57, 7-19, 7-53
copy() 6-16, 7-19
CPU core affinity 17-7
CPU usage 10-24
create_datawriter() 6-41
create_publisher() 6-7
create_topic() 5-4

D

data samples. See samples.
data types 2-4
  supported 3-4, 3-7
DATA_AVAILABLE status 4-19, 7-24, 7-36
DATA_ON_READERS status 4-19, 7-24
DATA_READER_CACHE status 7-36

DATA_READER_PROTOCOL status 7-37
DATA_WRITER_CACHE status 6-29
DATA_WRITER_PROTOCOL status 6-30
database cleanup thread 17-1 *to* 17-2
DATABASE message category 18-5
Database QoS 8-38
data-centric communications 2-2
DataReaderProtocol QoS 7-72
DataReaderResouceLimits QoS 10-15
DataReaderResourceLimits QoS 7-78
DataReaders
  checking status for 7-34
  copying QoS for 7-53
  copying Topic QoS 7-19
  creating/deleting 7-29
  finding matching writers 7-53
  Listeners for 7-31
  memory allocation 7-84
  operations on 7-25
  QoS for 7-47
  status for 7-35
DataWriterProtocol QoS 6-92, 10-18
DataWriterResourceLimits QoS 6-104
DataWriters
  copying Topic QoS 6-15, 6-57, 7-19
  creating/deleting 6-24
  definition 6-3, 7-5
  finding matching readers 6-59
  Listener's relationship to Publishers' 6-16
  Listeners for 6-26
  operations on 6-21
  ordering samples from multiple 6-110
  preventing starvation 7-84
  QoS for 6-49, 6-85
  resource limits for 6-104
  samples per 7-82
  saving samples for later use 6-113
  status for 6-20, 6-28 *to* 6-29
  writing data 6-41
DCPS 2-2, 2-11
DCPSParticipant 14-1
DCPSPublication 14-1
DCPSSubscription 14-1
DDS_BuiltinTopicReaderResourceLimits_t 8-50
DDS_DATAREADER_QOS_DEFAULT 7-29
DDS_DATAREADER_QOS_USE_TOPIC 7-53
DDS_DataReaderQos structure 7-47
DDS_DATAWRITER_QOS_USE_TOPIC_QOS 6-57
DDS_ParticipantBuiltinTopicData 14-3
DDS_ReliableWriterCacheEventCount 6-38
DDS_RtpsReliableReaderProtocol_t 7-76
DDS_RtpsReliableWriterProtocol_t 6-95
DDS_SubscriberQos structure 7-12
DDS_TransportMulticastSetting_t 7-91
deadline
  status for missing 6-36, 7-42
Deadline QoS 6-107
  interaction with TimeBasedFilter QoS 7-88
debugging error messages 12-28, 18-3, 18-6
default QoS 4-2
delete_contained_entities() 6-9, 7-12, 7-31, 8-18
depth (History QoS) 6-119
depth of queues 10-18
destination timestamp 6-110
DestinationOrder QoS 6-110
direct_communication 11-19
direct_communication (Durability QoS) 6-114
disable_fragmentation_support(DataReaderResourceLimits QoS) 7-79
disable_positive_acks (DataReaderProtocol QoS) 7-75
disable_positive_acks (DataWriterProtocol QoS) 6-94
disable_positive_acks_decrease_sample_keep_duration_factor (RtpsReliableWriterProtocol_t) 6-97
disable_positive_acks_enable_adaptive_sample_keep_duration (RtpsReliableWriterProtocol_t) 6-97
disable_positive_acks_increase_sample_keep_duration_factor (RtpsReliableWriterProtocol_t) 6-97
disable_positive_acks_max_sample_keep_duration (RtpsReliableWriterProtocol_t) 6-97
disable_positive_acks_min_sample_keep_duration (RtpsReliableWriterProtocol_t) 6-97
Discovery
  accessing Topics 8-33
  finding remote DomainParticipants 8-32
discovery
  debugging 12-28
  definition 12-2
  endpoint phase 12-3
  endpoint readers/writers 12-20
  entities used 12-10
  late-joiners 10-27
  participant phase 12-2
  ports used 8-70
  refresh mechanism 12-15
  related QoS 12-27
  summary diagram 12-26
Discovery QoS 8-42
DiscoveryConfig QoS 8-47
dispose() 6-47, 6-162
  vs. unregister_instance() 6-47
disposed_generation_count (DDS_SampleInfo) 7-67, 7-70
domain ID 8-19, 8-68
domain_id_gain 12-34
DomainParticipantFactory
  example code 8-4
  operations on 8-5, 13-4
  purpose of 8-4
  QoS for 8-7, 8-9
DomainParticipantResourceLimits QoS 8-54
DomainParticipants
  accessing discovered 8-32
  creating 8-16
  definition 8-2, 8-10
  deleting 8-18
  discovery of 8-50

domain IDs 8-19, 8-68
example code 6-8, 6-53, 7-50, 8-17, 8-21, 8-25
Listener of last resort 6-17, 7-21
Listeners for 8-20
operations on 8-11
QoS for 8-22
domains
definition 2-10, 8-1
multiple 2-11, 8-2
vs. partitions 6-72
dropped samples status 7-44
Durability QoS 6-113, 6-116, 10-27
dynamic memory 8-59
dynamically_allocate_fragmented_samples(DataReaderResourceLimits QoS) 7-81

E

enable() 4-3
enabled_transports 6-153, 8-42
enabling entities 6-64
recursiveness 4-4
end_coherent_changes() 6-44, 6-81
endpoint discovery 12-3
ENTITIES message category 18-5
EntityFactory QoS 4-4, 6-64
entityID 12-3
environment variables
NDDS_DISCOVERY_PEERS 12-4
NDDSHOME i-xxii
error messages 12-28, 18-3
format of 18-6
ERROR verbosity 18-4
Event QoS 8-61
event thread 17-1, 17-3
ExclusiveArea QoS 6-61, 6-66, 6-140
when to change 6-67
expects_inline_qos (DataReaderProtocol QoS) 7-75
expression parameters 5-21
external clock 8-75

F

factory class 4-3
fast_heartbeat_period (RtpsReliableWriterProtocol_t) 6-95
filter expression syntax 5-23
filter expressions 5-16, 5-21
finalize_instance() 8-9
finalizeX() 3-55
find_topic() 8-30
FlowControllers
creating and deleting 6-169
external trigger 6-171
flushing batched samples 6-43, 6-61, 6-87 *to* 6-88
FooDataReader
definition 7-55
operations on 7-26 *to* 7-27
FooDataWriter
definition 6-41
operations on 6-21
FooSeq 7-65
framing Heartbeat 10-34

G

generating code
See rtiddsgen.
generation_rank (DDS_SampleInfo) 7-67, 7-70 *to* 7-71
get_builtin_subscriber() 14-2
get_c_version() 18-2
get_core_api_version() 18-2
get_cpp_api_version() 18-2
get_datareader_cache_status() 7-34
get_datareader_protocol_status() 7-34
get_datareaders() 7-22
get_datawriter_cache_status() 6-28
get_datawriter_protocol_status() 6-28
get_deadline_missed_status() 6-36
get_default_datawriter_qos() 6-16, 7-19
get_discovered_participant_data() 8-32
get_discovered_participants() 8-32
get_discovered_topic_data() 8-33
get_discovered_topics() 8-33
get_domain_id() 8-20
get_instance_handle() 4-6, 8-33
get_key_value() 6-48, 7-54
get_listener() 4-7
get_liveliness_changed_status() 7-34, 7-37
get_liveliness_lost_status() 6-28, 6-36
get_matched_publication_data() 7-53
get_matched_publication_datareader_protocol_status() 7-34
get_matched_publications() 7-53 *to* 7-54
get_matched_subscription_data() 6-59
get_matched_subscription_datawriter_protocol_status_by_locator() 6-28
get_matched_subscription_datawriter_protocol_status() 6-28
get_matched_subscription_locators() 6-59
get_matched_subscriptions() 6-59
get_offered_deadline_missed_status() 6-28
get_offered_incompatible_qos_status() 6-28, 6-37
get_output_file() 18-5
get_participant() 6-18, 7-23
get_publication_match_status() 6-28
get_qos() 4-8, 6-52
get_reliable_reader_activity_changed_status() 6-28
get_reliable_writer_cache_changed_status() 6-28
get_requested_deadline_missed_status() 7-34, 7-42
get_requested_incompatible_qos_status() 7-34, 7-43
get_sample_lost_status() 7-34, 7-44
get_sample_rejected_status() 7-34, 7-44
get_status_changes() 4-6, 4-18, 4-21, 6-28, 7-34
get_statuscondition() 4-50, 7-34
get_subscription_match_status() 7-34, 7-46

get_trigger_value() 4-37
get_verbosity() 18-5
getting data 7-3 *to* 7-4, 7-31, 7-36, 7-57, 7-65
GroupData QoS 6-69
GUID (Globally Unique ID) 6-133, 8-70, 12-3

H

hash table 3-77, 6-94, 6-147, 8-55, 8-57
HB messages. See heartbeats.
heartbeat_period 10-19
  diagram 10-20
heartbeat_period (RtpsReliableWriterProtocol_t) 6-95, 6-99
heartbeat_suppression_duration (DDS_RtpsReliableReaderProtocol_t) 7-76
Heartbeats
  types of 10-34
heartbeats 10-5
  controlling 10-18
  how many resent 10-22
  how often 10-19
  response delays 7-76
heartbeats_per_max_samples 10-21
  diagram 10-23
heartbeats_per_max_samples (RtpsReliableWriterProtocol_t) 6-96
high_watermark (RtpsReliableWriterProtocol_t) 6-95
historical data 6-113, 7-35, 10-28
HISTORY QoS 10-18
History QoS 6-118
  depth 10-18
  effect of ResourceLimits QoS 6-120
history_depth (DurabilityService QoS) 6-117
history_kind (DurabilityService QoS) 6-117
host ID 8-70

I

IDL 2-4, 3-52
  including other files 3-76
  supported types 3-57, 3-63, 3-66
  unsupported types 3-53
ignore_participant() 14-13
ignore_publication() 7-54, 14-13
ignore_publisher() 6-70, 6-160
ignore_subscription() 6-59, 14-13
ignore_topic() 8-33, 14-13
implicit Publishers 6-6, 7-9
inactivate_nonprogressing_readers (RtpsReliableWriterProtocol_t) 6-95
incompatible QoS
  status for 6-37
info units 7-79
inheriting QoS profiles 15-20
initial and maximum values in QoS 7-78, 7-84
initial peers list 12-4
initial_batches (DataWriterResouceLimits QoS) 6-105
initial_concurrent_blocking_threads (DataWriterResourceLimits QoS) 6-105
initial_fragmented_samples(DataReaderResourceLimits QoS) 7-80
initial_infos (DataReaderResourceLimits QoS) 7-79
initial_instances (ResourceLimits QoS) 6-147
initial_outstanding_reads (DataReaderResourceLimits QoS) 7-79
initial_participant_announcements (DiscoveryConfig QoS) 8-48
initial_peers 8-42
  adding to 8-43
initial_records 8-39
initial_remote_virtual_writers_per_instance(DataReaderResourceLimits QoS) 7-82
initial_remote_writers (DataReaderResourceLimits QoS) 7-79
initial_remote_writers_per_instance (DataReaderResourceLimits QoS) 7-79
initial_samples (ResourceLimits QoS) 6-147
initial_weak_references 8-40
instance handle 4-6
instance ID 8-70
instance state 7-69
  interaction with Ownership QoS 7-69
instance states 6-48
instance_handle (DDS_SampleInfo) 7-67
instance_hash_buckets (ResourceLimits QoS) 6-147
instance_replacement (DataWriterResouceLimits QoS) 6-105
instance_state (DDS_SampleInfo) 7-67
instances
  registration 6-45
  registration example 6-47
Interface Description Language. See IDL
internal clock 8-75
inter-participant reader/writer 12-18

K

keep duration 6-100, 10-26
key hash 6-94, 6-147, 8-55, 8-57
key-hash 3-77, 6-94
keys
  definition 6-45
  getting value of 6-48
  in IDL file 3-77
  managing data instances 6-45
  registering instances 6-45
  rtiddsgen 3-77

L

last_reason (SAMPLE_REJECTED status) 7-45
late_joiner_heartbeat_period (RtpsReliableWriterProtocol_t) 6-95
late-joiners 10-27
  discovery of 12-24

latency 7-77
layer 2 switches 16-5
lease_duration (Liveliness QoS) 6-126
LENGTH_UNLIMITED 8-40
Listeners
basic steps 7-3
creating and deleting 4-29
definition 6-16
example code 6-19, 7-22, 7-33, 8-21
for DataReaders 7-31
for DataWriters 6-26
for DomainParticipants 8-20
for Publishers 6-16
for Topics 5-10
last resort 6-17, 7-21, 8-20
nil (or no-op) 4-29
operations allowed in 4-32
precedence of 6-17, 6-27, 7-21
purpose of 4-19
relationship of PublisherListener and DataWriterListener 6-16
removing 8-20
listening for data 7-3
liveliness assertion during write() 6-42
LIVELINESS_CHANGED status 6-39, 7-37
LIVELINESS_LOST status 6-36
locators (TransportUnicast QoS) 6-154
logged error messages 18-3
long double 3-73 *to* 3-74
lookup_datareader() 6-18, 7-23, 14-2, 14-12
lookup_instance() 6-48, 7-54
lookup_participant() 8-9
lookup_topicdescription() 8-29 *to* 8-30
lost samples 7-44
low_watermark (RtpsReliableWriterProtocol_t) 6-95
low-bandwidth connections 7-89

M

matching writers and readers 6-37, 6-59, 7-42, 7-53, 12-3
status for 7-46
max_batches (DataWriterResouceLimits QoS) 6-105
max_blocking_time (Reliability QoS) 6-106, 6-143, 10-10
max_bytes_per_nack_response (RtpsReliableWriterProtocol_t) 6-97, 10-22
max_concurrent_blocking_threads (DataWriterResourceLimits QoS) 6-105
max_data_bytes (Batch QoS) 6-86
max_flush_delay (Batch QoS) 6-86
max_fragmented_samples_per_remote_writer(DataReaderResourceLimits QoS) 7-80
max_fragmented_samples(DataReaderResourceLimits QoS) 7-80
max_fragments_per_sample(DataReaderResourceLimits QoS) 7-80
max_heartbeat_response_delay (RtpsReliableReaderProtocol_t) 7-76
max_heartbeat_response_delay(DDS_RtpsReliableReaderProtocol_t) 7-76
max_heartbeat_retries (RtpsReliableWriterProtocol_t) 6-95, 10-22
max_infos (DataReaderResourceLimits QoS) 7-79
max_instances (DurabilityService QoS) 6-117
max_instances (ResourceLimits QoS) 6-147, 7-83
max_liveliness_loss_detection_period (DiscoveryConfig QoS) 8-48
max_nack_response_delay (DataWriterProtocol QoS) 10-24
max_nack_response_delay (RtpsReliableWriterProtocol_t) 6-96
max_outstanding_reads (DataReaderResourceLimits QoS) 7-79
max_remote_reader_filters (DataWriterResouceLimits QoS) 6-105
max_remote_virtual_writers_per_instance(DataReaderResourceLimits QoS) 7-81
max_remote_writers (DataReaderResourceLimits QoS) 7-79
max_remote_writers_per_instance (DataReaderResourceLimits QoS) 7-78 *to* 7-79
max_samples (Batch QoS) 6-86
max_samples (DurabilityService QoS) 6-117
max_samples (ResourceLimits QoS) 6-38, 6-147, 10-21
max_samples_per_instance (DurabilityService QoS) 6-117
max_samples_per_instance (ResourceLimits QoS) 6-147, 10-18
max_samples_per_read(DataReaderResourceLimits QoS) 7-79
max_samples_per_remote_writer 10-15
max_samples_per_remote_writer (DataReaderResourceLimits QoS) 7-79, 7-82 *to* 7-83
max_send_window_size (RtpsReliableWriterProtocol_t) 6-98
max_skiplist_level 8-39
max_total_instances (DataReaderResourceLimits QoS) 7-81, 7-83
max_weak_references 8-40
maximizing throughput 7-77
memory
allocation 3-54, 7-78, 7-84, 8-58
clearing 6-16, 7-19
copying 7-19
returning 8-9
message bundling 10-5
message storms 10-27
meta data
ports for 12-32
meta-traffic 8-42
definition 17-5
metatraffic_transport_priority 8-42, 8-45
min_heartbeat_response_delay (RtpsReliableReaderProtocol_t) 7-76

min_nack_response_delay
(RtpsReliableWriterProtocol_t) 6-96
min_send_window_size
(RtpsReliableWriterProtocol_t) 6-98
minimizing latency 7-77
minimum_separation
reasons for changing 7-89
minimum_separation (TimeBasedFilter QoS) 7-87, 7-89
missed deadline status 6-36, 7-42
missing samples status 7-44
module (IDL type) 3-61, 3-66, 3-72
monotonic clock 8-75
multicast
addresses 7-90, 8-44
example code 7-92
locators 7-90
ports 7-90
ports used 12-32
TransportMulticast QoS 7-90
multicast_receive_addresses 8-42, 8-44

N

nack_suppression_duration
(RtpsReliableWriterProtocol_t) 6-97
NACKs
non-progressing 10-24
namespace (rtiddsgen option) 3-113
NDDS_Config_LibraryVersion_t structure 18-3
NDDS_DISCOVERY_PEERS 12-4
NDDSConfigLogger class 18-5
operations on 18-5
NDDSConfigVersion class 18-2
NDDSHOME i-xxii
NEW view state 7-68
new_remote_participant_announcement_period
(DiscoveryConfig QoS) 8-48
nil listener 4-29
no_writers_generation_count (DDS_SampleInfo) 7-67, 7-70
non-progressing NACK 10-24
NOT_ALIVE_DISPOSED instance state 7-69
NOT_ALIVE_NO_WRITERS instance state 6-48, 7-69
NOT_NEW view state 7-68
NOT_READ sample state 7-68
notification of new data 7-31

O

object-centric communications 2-2
offered QoS 6-51
OFFERED_DEADLINE_MISSED status 6-36
OFFERED_INCOMPATIBLE_QOS status 6-37
on_data_available() 7-32, 7-36
on_data_on_readers() 7-24
on_instance_replaced() 6-27
on_liveliness_changed() 7-32, 7-37
on_liveliness_lost() 6-27, 6-36
on_offered_deadline_missed() 6-27, 6-36
example 6-19
on_offered_incompatible_qos() 6-27, 6-37
on_publication_matched() 6-27
on_reliable_reader_activity_changed() 6-27
on_reliable_writer_cache_changed() 6-27
on_requested_deadline_missed() 7-32, 7-42
on_requested_incompatible_qos() 7-32, 7-43
on_sample_lost() 7-32, 7-44
on_sample_rejected() 7-32, 7-44
on_subscription_matched() 7-32, 7-46
order of samples 6-110
ordered_access (Presentation QoS) 6-80 *to* 6-81
effect of 6-83
ownership of data 7-57
effect of unregistering 6-46
Ownership QoS
effect of sharing 7-84
effect on instance state 7-69
preventing starvation 7-84
OwnershipStrength QoS 6-136

P

packet loss 10-14
participant DATA messages 12-2, 12-12, 12-15
participant ID 8-68 *to* 8-69
participant_assert_liveliness_period 12-12
participant_id (WireProtocol QoS) 8-67
participant_id_gain 12-34
participant_liveliness_assert_period 12-12
participant_liveliness_assert_period (DiscoveryConfig QoS) 8-47
participant_liveliness_assert_period (DisocveryConfig QoS) 8-50, 8-53
participant_liveliness_lease_duration
(DiscoveryConfig QoS) 8-47
participant_liveliness_lease_duration_period
(DiscoveryConfig QoS) 12-12
participant_reader_resource_limits (DiscoveryConfig QoS) 8-48
ParticipantBuiltinTopicData 14-14
Partition QoS 6-72
example 6-76
example of changing 6-76
impact on memory 6-79
partitions
definition 6-72
rules for matching names 6-74
vs. domains 6-72
wildcards 6-74
peers list 12-2, 12-4
adding to 8-43
peer-to-peer communication 11-19
performance
improving with registration 6-45
period (Deadline QoS) 6-107

piggyback heartbeats 10-21
plain communication status 4-19, 7-24
PLATFORM message category 18-5
platforms supported i-xxii
plugin_data 6-158
pointer (IDL type) 3-58, 3-68
point-to-point communication 1-3
polling for data 7-3 *to* 7-4
port numbers 8-68
ports 8-70, 12-32 *to* 12-33
preprocessor directives 3-76, 3-114
Presentation QoS 6-79
preventing starvation 7-84
promiscuous mode 7-93
Property QoS 6-137, 8-34
PRSTDataReader 11-19, 22-1, 22-13
PRSTDataWriter 11-19, 22-1, 22-13
publication DATA messages 12-3
publication_handle (DDS_SampleInfo) 7-67
PUBLICATION_MATCHED status 6-37
publication_reader (DiscoveryConfig QoS) 8-49
publication_reader_ resource_limits 8-48
publication_squence_number (DDS_SampleInfo) 7-67
publication_writer 8-49
publications 2-9
  definition 2-9
Publishers
  creating 6-7
  definition 2-8, 6-3
  deleting 6-9, 8-18
  example Listener 6-19
  implicit 6-6, 7-9
  Listener's relationship to DataWriters' Listener 6-16
  Listeners for 6-16
  operations on 5-20, 6-5, 6-165
  QoS for 6-10, 6-60
  setting QoS for 6-9
publish-subscribe communications 1-4, 2-8
pulled samples 6-30
push_on_write (DataWriterProtocol QoS) 6-93
  when to change 6-103
pushed samples 6-30

Q

QoS
  changing defaults 4-2
  compatibility 6-37, 6-51, 7-47
  Database 8-38
  DataReaderProtocol 7-72
  DataReaderResourceLimits 7-78
  DataReaders 7-47
  DataWriterProtocol 6-92, 10-18
  DataWriterResourceLimits 6-104
  Deadline 6-107
  defaults 4-2
  DestinationOrder 6-110
  Discovery 8-42
  DiscoveryConfig 8-47
  DomainParticipantResourceLimits 8-54
  Durability 6-113, 6-116, 10-27
  EntityFactory 6-64
  Event 8-61
  ExclusiveArea 6-61, 6-66, 6-140
  for built-in Subscribers and DataReaders 14-10
  for DataWriters 6-49, 6-85
  for DomainParticipantFactory 8-7, 8-9
  for DomainParticipants 8-22
  for Publishers 6-60
  for Topics 5-12
  GroupData 6-69
  History 6-118, 10-18
  how to set 4-8
  offered 6-51
  OwnershipStrength 6-136
  Partition 6-72
  Property 6-137, 8-34
  Publishers 6-9
  ReceiverPool 8-64
  Reliability 6-142
  requested 6-51
  ResourceLimits 6-146
  rules for setting 4-9
  Subscribers 7-12, 7-72
  summary table 4-13
  TimeBasedFilter 7-87
  TransportBuiltin 8-66
  TransportMulticast 7-90
  TransportPriority 6-151
  TransportSelection 6-152
  TransportUnicast 6-154
  TypeSupport 6-157
  UserData 6-158
  using defaults 7-29
  using Topic's 7-53
  WireProtocol 8-67
QoS. See Also individual QoS policy names.
Quality of Service. See QoS
queue depths 10-18

R

read communication status 4-19, 7-24, 7-36
READ sample state 7-68
read_instance() 7-54, 7-61, 7-63
read_next_instance() 7-62
read_next_sample() 7-61
read() vs. take() 7-57
reading data 7-57
real-time applications
  features of 2-2
realtime clock 8-75
receive queue 10-2
  example 10-17
  size 6-147, 10-11, 10-13, 10-15, 10-18
receive thread 8-64, 17-1, 17-4

receive_address (DDS_TransportMulticastSetting_t) 7-91
receive_port (DDS_TransportMulticastSetting_t) 7-91
receive_port (TransportUnicast QoS) 6-155
ReceiverPool QoS 8-64
receiving data 7-31, 7-36, 7-57, 7-65
  basic steps 7-3 *to* 7-4
reception_timestamp (DDS_SampleInfo) 7-67
refilter (History QoS) 6-119
register_instance() 6-45 *to* 6-46, 6-48
rejected samples 7-45
relay communication 11-19
Reliability QoS 6-142
  effect on bandwidth 6-146
reliable delivery 6-142, 10-2
  blocking time 10-10
  definition 10-2
  diagram 10-7
  strict 10-18
  tuning 10-9
RELIABLE_READER_ACTIVITY_CHANGED status 6-39
RELIABLE_WRITER_CACHE_CHANGED status 6-38
remote_participant_purge_kind (DiscoveryConfig QoS) 8-48
repair packages 10-22
replace_empty_instances (DataWriterResouceLimits QoS) 6-105
requested QoS 6-51
REQUESTED_DEADLINE_MISSED status 7-42
REQUESTED_INCOMPATIBLE_QOS status 7-43
resource configuration 7-78
resource limits 7-78
ResourceLimits QoS 6-146
  effect on History QoS 6-120
response delays 7-76
restricting communication 14-13
retries
  controlling 10-18
RTI Data Distribution Service version query 18-1
rtiddsgen
  @copy directive 3-78
  @key directive 3-77
  @resolve-name directive 3-79
  @top-level directive 3-83
  command-line options 3-111
  directives 3-76
  generated files 3-108 *to* 3-109
  IDL conversions 3-57, 3-63, 3-66
  including IDL files 3-76
  support types 3-57, 3-63, 3-66
  syntax 3-111
RTPS protocol 8-67, 10-4
  basic behavior diagram 10-6
  bundling messages 10-5
  reliable behavior diagram 10-7
rtps_app_id 12-27
rtps_app_id (WireProtocol QoS) 8-68
rtps_auto_id_kind (WireProtocol QoS) 8-68
rtps_host_id 12-27
rtps_host_id (WireProtocol QoS) 8-68
rtps_instance_id (WireProtocol QoS) 8-68
rtps_object_id (DataReaderProtocol QoS) 7-74
rtps_object_id (DataWriterProtocol QoS) 6-93
rtps_reliable_reader (DataReaderProtocol QoS) 7-76
rtps_reliable_writer (DataWriterProtocol QoS) 6-94
rtps_reserved_ports_mask (WireProtocol QoS) 8-68
rtps_well_known_ports (WireProtocol QoS) 8-68
RxO 6-51

S

sample state 7-68
SAMPLE_LOST status 7-44
sample_rank (DDS_SampleInfo) 7-70 *to* 7-71
sample_rank (SampleInfo) 7-67
SAMPLE_REJECTED status
  reason codes 7-45
sample_state (DDS_SampleInfo) 7-67
SampleInfo structure 7-67
samples
  coherent sets 6-44
  count of lost 7-44
  definition 2-5, 2-7
  how many per DataWriter 7-82
  lost 7-44
  notification of arrival 7-36
  order of delivery 6-110
  ownership of data 7-57
  reasons for rejection 7-45
  resending 10-22, 10-24
  saving for later use 6-113
  structure of 7-67
  unacknowledged 6-38
  writing 6-41
send queue
  blocking time 10-10
  max_samples 10-13
  purpose of 10-2
  size 6-119, 6-147, 10-11 *to* 10-12, 10-14 *to* 10-15, 10-18
  size (formula) 10-13 *to* 10-14
  unblocking when full 6-106
send_window_decrease_factor (RtpsReliableWriterProtocol_t) 6-98
send_window_increase_factor (RtpsReliableWriterProtocol_t) 6-98
send_window_update_period (RtpsReliableWriterProtocol_t) 6-98
sending data. See writing data.
sequence data type 3-54, 7-65
sequence number 7-67
sequences (defined) 7-65
service_cleanup_delay (DurabilityService QoS) 6-117
set_default_datawriter_qos() 4-9

set_enabled_statuses() 4-49
set_listener() 4-7
set_output_file() 18-5
set_qos() 4-8, 6-52
set_verbosity() 18-5
shared ownership
  effect of max_samples_per_remote_writer 7-83
shutdown_cleanup_period 8-39
shutdown_timeout 8-39
SILENT verbosity 18-4
skip list 8-39
source timestamp 6-110
source_timestamp (DDS_SampleInfo) 7-67
source_timestamp_resolution (Batch QoS) 6-87
speculative caching 10-41
SQL syntax 5-23
starvation 7-83
static memory allocation 8-59
status changes 7-34
STATUS_ALL verbosity 18-4
STATUS_LOCAL verbosity 18-4
STATUS_REMOTE verbosity 18-4
statuses
  changes in 4-22
  DATA_AVAILABLE 4-19, 7-24, 7-36
  DATA_ON_READERS 4-19, 7-24
  DATA_READER_CACHE 7-36
  DATA_READER_PROTOCOL 7-37
  DATA_WRITER_CACHE 6-29
  DATA_WRITER_PROTOCOL 6-30
  for DataReaders 7-34 *to* 7-35
  for DataWriters 6-20, 6-28 *to* 6-29
  for Subscribers 7-24
  LIVELINESS_CHANGED 7-37
  LIVELINESS_LOST 6-36
  OFFERED_DEADLINE_MISSED 6-36
  OFFERED_INCOMPATIBLE_QOS 6-37
  plain communication status 4-19
  PUBLICATION_MATCHED 6-37
  read communication status 4-19
  RELIABLE_READER_ACTIVITY_CHANGED 6-39
  RELIABLE_WRITER_CACHE_CHANGED 6-38
  REQUESTED_DEADLINE_MISSED 7-42
  REQUESTED_INCOMPATIBLE_QOS 7-43
  SAMPLE_LOST 7-44
  SUBSCRIPTION_MATCHED 7-46
stock quote example 2-5 *to* 2-6
string data type 3-6
submessage bundling 10-5
Subscribers
  creating 7-10
  definition 2-9, 7-5
  operations on 7-7
  QoS for 7-12, 7-72
  statuses for 7-24
SUBSCRIPTION_MATCHED status 7-46
subscription_reader (DiscoveryConfig QoS) 8-49
subscription_reader_resource_limits (DiscoveryConfig QoS) 8-48
subscription_writer (DiscoveryConfig QoS) 8-49
subscriptions
  notification timing 7-88
supported data types 3-4, 3-7

T

take_instance() 7-54
take() vs. read() 7-57
taking data 7-57
thread_safe_write (Batch QoS) 6-87
thread-pinning 17-7
threads 6-105
  receive 8-64
  shared/exclusive areas for 6-61, 6-66, 6-140
  unblocking 6-106
throughput 7-77
TimeBasedFilter QoS 7-87
  interaction with Deadline QoS 7-88
timestamp 6-42, 6-110
timestamp resolution 6-87, 6-90
timestamp tolerance 6-111
timestamps 8-75
to_string() 18-2
topic_name
  definition 5-4
Topics 2-9
  accessing discovered 8-33
  copying QoS 6-15, 7-19
  creating 5-4
  definition 2-5, 2-7
  example code 6-12, 6-53 *to* 6-54, 7-14, 7-51, 8-26
  Listeners for 5-10
  operations on 5-3
  purpose of 5-2
  purpose of QoS 5-12
  QoS for 5-6, 5-12
  waiting for creation 8-30
TRANSIENT_LOCAL (Durability QoS) 6-113 *to* 6-114
transport plug-ins
  for meta-traffic 8-42
TransportBuiltin QoS 8-66
TransportMulticast QoS 7-90
TransportPriority QoS 6-151
transports (TransportMulticast QoS) 7-91
transports (TransportUnicast QoS) 6-155
TransportSelection QoS 6-152
TransportUnicast QoS 6-154
trigger_value 4-37, 4-49
type_code_max_serialized_length (DomainParticipantsResourceLimits QoS) 8-57
type_name
  definition 5-4
typedef construct 3-74
TypeSupport QoS 6-157
typographical conventions i-xxii

U

unacknowledged samples 6-38
unicast 6-154
  ports used 12-33
unregister_instance() 6-46
  vs. dispose() 6-47
unregister() 6-162
unregistering instances
  instance ownership after 6-46
unsigned types 3-74
use_shared_exclusive_area (ExclusiveArea QoS) 6-67
user data
  ports for 12-33
user data types
  definition 3-4
  See also data types.
UserData QoS 6-158

V

valid_data (DDS_SampleInfo) 7-67
verbosity 18-3
version of RTI Data Distribution Service 18-1
view state 7-68
view_state (DDS_SampleInfo) 7-67
virtual GUID 11-6
virtual_guid (DataReaderProtocol QoS) 7-74
virtual_guid (DataWriterProtocol QoS) 6-93
VOLATILE (Durability QoS) 6-113 *to* 6-114

W

wait_for_acknowledgments() 6-18, 6-44
wait_for_historical_data() 7-35
waiting for data 7-3
WaitSets 4-37
  creating and deleting 4-38
  example code 4-44
  operations on 4-41
  properties for 4-38
  purpose of 4-37
WARNING verbosity 18-4
wchar 3-74
weak references 8-40
well_known_ports 8-70
wildcard partition names 6-74
WireProtocol QoS 8-67
write_w_timestamp() 6-42
writing data 6-41
  basic steps 6-1
  blocked send queue 10-11
  example with registration 6-47
  liveliness assertion 6-42
  registration of instance 6-45