Find an updated version of this guide at https://community.rti.com/documentation.

RTI Connext DDS

Core Libraries

Getting Started Guide

Version 6.0.1

© 2020 Real-Time Innovations, Inc.
All rights reserved.
Printed in U.S.A. First printing.
March 2020.

Trademarks

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

Copy and Use Restrictions

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

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

The security features of this product include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

Technical Support

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

Contents

Chapter 1 Welcome to RTI Connext DDS!

1.1 A Guide to the Provided Documentation 2

1.2 Paths Mentioned in Documentation 4

1.3 Why Choose Connext DDS? 5

1.3.1 Reduce Risk Through Performance and Availability 5

1.3.2 Reduce Cost through Ease of Use and Simplified Deployment 6

1.3.3 Ensure Success with Unmatched Flexibility 6

1.3.4 Connect Heterogeneous Systems 7

1.3.5 Interoperate with Databases, Event Engines, and JMS Systems 7

1.4 What Can Connext DDS Do? 7

1.5 Am I Better Off Building My Own Middleware? 9

Chapter 2 Navigating the Directories 10

Chapter 3 Selecting a Development Environment

3.1 Using the Command-Line Tools 12

3.2 Using Microsoft Visual Studio 13

Chapter 4 A Quick Overview

4.1 Building and Running “Hello, World” 14

4.1.1 Step 1: Set Up the Environment 14

4.1.1.1 Set Up the Environment on Your Development Machine 14

4.1.1.2 Set Up the Environment on Your Deployment Machine 17

4.1.2 Step 2: Compile the Hello World Program 18

4.1.3 Step 3: Start the Subscriber 21

4.1.4 Step 4: Start the Publisher 22

4.2 Building and Running a Request-Reply Example 24

4.3 An Introduction to DDS 25

4.3.1 An Overview of DDS Objects 26

4.3.2 DomainParticipants 26
4.3.2.1 What is QoS? 29
4.3.3 Publishers and DataWriters 29
4.3.4 Subscribers and DataReaders 31
4.3.5 Topics 35
4.3.6 Keys and DDS Samples 36
4.3.7 Requesters and Repliers 37
Chapter 5 Capabilities and Performance
5.1 Automatic Application Discovery 40
5.1.1 When to Set the Discovery Peers 40
5.1.2 How to Set Your Discovery Peers 41
5.2 Customizing Behavior: QoS Configuration 42
5.3 Compact, Type-Safe Data Programming with DDS Data Types 45
5.3.1 Using Built-in DDS Types 48
5.3.2 Using DDS Types Defined at Compile Time 48
5.3.3 Generating Code with RTI Code Generator 49
5.3.3.1 Building the Generated Code 52
5.3.3.2 Running the Example Applications 54
5.3.4 Running with Dynamic DDS Types 56
Chapter 6 Design Patterns for Rapid Development
6.1 Building and Running the News Examples 58
6.2 Subscribing Only to Relevant Data 60
6.2.1 Content-Based Filtering 61
6.2.1.1 Implementation 61
6.2.1.2 Running & Verifying 62
6.2.2 Lifespan and History Depth 63
6.2.2.1 Implementation 63
6.2.2.2 Running & Verifying 65
6.2.3 Time-Based Filtering 67
6.2.3.1 Implementation 68
6.2.3.2 Running & Verifying 69
6.3 Accessing Historical Data when Joining the Network 69
6.3.1 Implementation 70
6.3.2 Running & Verifying 71
6.4 Caching Data within the Middleware 71
6.4.1 Implementation 72

6.4.2 Running & Verifying 75
6.5 Receiving Notifications When Data Delivery Is Late 75
6.5.1 Implementation 75
6.5.1.1 Offered Deadlines 75
6.5.1.2 Requested Deadlines 76
6.5.2 Running & Verifying 78
Chapter 7 Design Patterns for High Performance
7.1 Building and Running the Code Examples 81
7.1.1 Understanding the Performance Results 82
7.1.2 Is this the Best Possible Performance? 84
7.2 Reliable Messaging 84
7.2.1 Implementation 85
7.2.1.1 Enable Reliable Communication 85
7.2.1.2 Set History To KEEP_ALL 86
7.2.1.3 Controlling Middleware Resources 87
7.3 High Throughput for Streaming Data 88
7.3.1 Implementation 90
7.4 Sending Large Data 91
7.4.1 FlatData Language Binding 91
7.4.2 Zero Copy Transfer Over Shared Memory 93
7.4.3 Choosing between FlatData Language Binding and Zero Copy Transfer over Shared Memory 94
7.5 Streaming Data over Unreliable Network Connections 95
7.5.1 Implementation 96
7.5.1.1 Managing Your Sample Size 96
7.5.1.2 Acknowledge and Repair Efficiently 98
7.5.1.3 Make Sure Repair Packets Don’t Exceed Bandwidth Limitation 99
7.5.1.4 Use Batching to Maximize Throughput for Small Samples 100

Chapter 1 Welcome to RTI Connext DDS!

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

- Ultra-low latency, extremely-high throughput messaging
- Industry-leading reliability and determinism
- Connectivity for heterogeneous systems spanning thousands of applications

Connext DDS is flexible; extensive quality-of-service (QoS) parameters adapt to your application, assuring that you meet your real-time, reliability, and resource usage requirements.

This document introduces basic concepts and summarizes how *Connext DDS* addresses your high-performance needs. After this introduction, we'll jump right into building distributed systems. The rest of this guide covers:

- First steps: Creating your first simple application.
- Learning more: An overview of the APIs and programming model with a special focus on the communication model, data types and qualities of service.
- Towards real-world applications: An introduction to meeting common real-world requirements.

To install *Connext DDS*, see the separate *RTI Connext DDS Installation Guide*.

1.1 A Guide to the Provided Documentation

We invite you to explore further by referring to the wealth of available information, examples, and resources.

> We refer to the main installation directory as **<NDDSHOME>**. See 1.2 Paths Mentioned in Documentation on page 4.

After installing *Connext DDS*, you'll find these in **<NDDSHOME>/doc/manuals/connext_dds**:

- **Installation Guide**—Describes how to install *Connext DDS* and license management.
- **Getting Started Guide**—Introduces you to the benefits and concepts behind the product and takes you step-by-step through the creation of a simple example application.

  If you want to use the *Connext DDS* Extensible Types feature, please also read:

  - **Addendum for Extensible Types** —Extensible Types allow you to define DDS data types in a more flexible way. Your DDS data types can evolve over time—without giving up portability, interoperability, or the expressiveness of the DDS type system.

  If you are using *Connext DDS* on an Android®, iOS®, or embedded platform, or with a database, you will find additional documents that specifically address these configurations:

  - **Addendum for Android Systems**
  - **Addendum for iOS Systems**
  - **Addendum for Embedded Systems**
  - **Addendum for Database Setup**

- **What's New**—Provides an overview of new features and enhancements in the current version of *Connext DDS*.
- **Release Notes**—Describes system requirements, what's new, what's fixed, and known issues.
- **User's Manual**—Describes the features of the product and how to use them. It is organized around the structure of the *Connext DDS* APIs and certain common high-level tasks.
- **Platform Notes**—Provides platform-specific information about the product, including specific information required to build your applications using *Connext DDS*, such as compiler flags and libraries.
- **API Reference HTML Documentation (<NDDSHOME>/README.html)**—This extensively cross-referenced documentation, available for all supported programming languages, is your in-depth reference to every operation and configuration parameter in *Connext DDS*. Even experienced *Connext DDS* developers will often consult this information.
- **The Programming How To's** provide a good place to begin learning the APIs. These are hyperlinked code snippets to the full API documentation.

From the **README.html** file, select one of the supported programming languages, then scroll down to the Programming How To's. Start by reviewing the Publication Example and Subscription Example, which provide step-by step examples of how to send and receive data with *Connext DDS*.

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

- **The Migration Guide** describes how to migrate to the current release from a previous *Connext DDS* release, including what compatibility issues you may need to account for during your upgrade. It is provided on the RTI Community Portal (https://community.rti.com/documentation) and is updated as needed.
- **The RTI Customer Portal** (http://support.rti.com/) Use this portal to download RTI software, access documentation and contact RTI Support. The RTI Customer Portal requires a username and password. You will receive this in the email confirming your purchase. If you do not have this email, please contact **license@rti.com**. Resetting your login password can be done directly at the RTI Customer Portal.
- **The RTI Community website** (https://community.rti.com) provides a wealth of knowledge to help you use *Connext DDS*, including:
  - Documentation, at https://community.rti.com/documentation
  - Best Practices
  - Example code for specific features, as well as more complete use-case examples,
  - Solutions to common questions,
  - A glossary,
  - Downloads of experimental software,
  - And more.
- Whitepapers and other articles are available from http://www.rti.com/resources

Of course, RTI also offers excellent technical support and professional services. To contact RTI Support, simply log into the Customer Portal, send email to support@rti.com, or call the telephone number provided for your region.

1.2 Paths Mentioned in Documentation

The documentation refers to:

- **<NDDSHOME>**

  This refers to the installation directory for *RTI® Connext® DDS*. The default installation paths are:

  - macOS® systems:
    /Applications/rti_connext_dds-6.0.1
  - Linux systems, non-*root* user:
    /home/<*your user name*>/rti_connext_dds-6.0.1
  - Linux systems, *root* user:
    /opt/rti_connext_dds-6.0.1
  - Windows® systems, user without Administrator privileges:
    <*your home directory*>\rti_connext_dds-6.0.1
  - Windows systems, user with Administrator privileges:
    C:\Program Files\rti_connext_dds-6.0.1

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

  Wherever you see **<NDDSHOME>** used in a path, replace it with your installation path.

  Note for Windows Users: When using a command prompt to enter a command that includes the path **C:\Program Files** (or any directory name that has a space), enclose the path in quotation marks. For example:

  ```
  "C:\Program Files\rti_connext_dds-6.0.1\bin\rtiddsgen"
  ```

  Or if you have defined the **NDDSHOME** environment variable:

  ```
  "%NDDSHOME%\bin\rtiddsgen"
  ```

- ***<path to examples>***

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

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

  Default path to the examples:

  - macOS systems: **/Users/<*your user name*>/rti_workspace/6.0.1/examples**
  - Linux systems: **/home/<*your user name*>/rti_workspace/6.0.1/examples**

- Windows systems: **<*your Windows documents folder*>\rti_workspace\6.0.1\examples**

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

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

1.3 Why Choose Connext DDS?

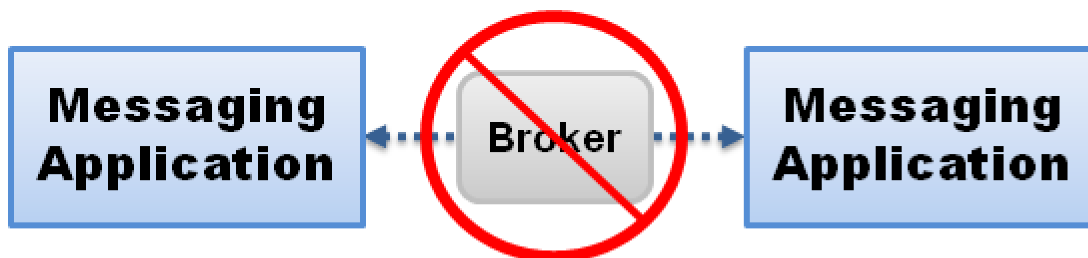
Connext DDS implements publish/subscribe networking for high-performance distributed applications. It complies with the Data Distribution Service (DDS) standard from the Object Management Group (OMG). Developers familiar with JMS and other middleware will see similarities, but will also find that DDS is not just another MOM (message-oriented middleware) standard! Its unique peer-to-peer architecture and unprecedented flexibility delivers much higher performance and adaptation to challenging applications. DDS is the first truly real-time networking technology standard. *Connext DDS* is by far the market leading implementation of DDS.

1.3.1 Reduce Risk Through Performance and Availability

Connext DDS provides top performance, whether measured in terms of latency, throughput, or real-time determinism[1]. One reason is its elegant peer-to-peer architecture.

Traditional messaging middleware requires dedicated servers to broker messages—introducing throughput and latency bottlenecks and timing variations. Brokers also increase system administration costs and represent single points of failure within a distributed application, putting data reliability and availability at risk.

Connext DDS doesn't use brokers. Messages flow directly from publishers to subscribers with minimal overhead. All the functions of the broker, including discovery (finding data sources and sinks), routing (directing data flow), and naming (identifying DDS data types and Topics) are handled in a fully-distributed, reliable fashion behind the scenes. It requires no special server software or hardware.



[1]You can review the data from several performance benchmarks here: http://www.rti.com/products/dds/benchmarks.html. Updated results for new releases are typically published within two months after general availability of that release.

Traditional message-oriented middleware implementations require a broker to forward every message, increasing latency and decreasing determinism and fault tolerance. RTI's unique peer-to-peer architecture eliminates bottlenecks and single points of failure.

The design also delivers high reliability and availability, with automatic failover, configurable retries, and support for redundant publishers, subscribers, networks, and more. Publishers and subscribers can start in any order, and enter and leave the network at any time; *Connext DDS* will connect and disconnect them automatically. *Connext DDS* provides fine-grained control over failure behavior and recovery, as well as detailed status notifications to allow applications to react to situations such as missed delivery deadlines, dropped connections, and slow or unresponsive nodes.

The *RTI Connext DDS Core Libraries User's Manual* has details on these and all other capabilities. This guide only provides an overview.

1.3.2 Reduce Cost through Ease of Use and Simplified Deployment

Connext DDS helps keep development and deployment costs low by:

- **Increasing developer productivity**—Easy-to-use, standards-compliant DDS APIs get your code running quickly. DDS is the established connectivity framework standard for real-time publish/subscribe communication in the defense industry and is expanding rapidly in utilities, transportation, intelligence, finance, and other commercial industries.
- **Simplifying deployment**—*Connext DDS* automatically discovers connections, so you don't need to configure or manage server machines or processes. This translates into faster turnaround and lower overhead for your deployment and administration needs.
- **Reducing hardware costs**—Traditional messaging products require dedicated servers or acceleration hardware in order to host brokers. The extreme efficiency and reduced overhead of RTI's implementation, on the other hand, allows you to achieve the same performance using standard off-the-shelf hardware, with fewer machines than the competition.

1.3.3 Ensure Success with Unmatched Flexibility

Out of the box, *Connext DDS* is configured to achieve simple data communications. However, when you need it, RTI provides a high degree of fine-grained, low-level control over the middleware, including:

- The volume of meta-traffic sent to assure reliability.
- The frequencies and time-outs associated with all events within the middleware.
- The amount of memory consumed, including the policies under which additional memory may be allocated by the middleware.

RTI's unique and powerful Quality-of-Service (QoS) policies can be specified in configuration files so that they can be tested and validated independently of the application logic. When not specified, *Connext DDS* will use default values chosen to provide good performance for a wide range of applications.

The result is simple-to-use networking that can expand and adapt to challenging applications, both current and future. RTI eliminates what is perhaps the greatest risk of commercial middleware: outgrowing the capability or flexibility of the design.

1.3.4 Connect Heterogeneous Systems

Connext DDS provides complete functionality and industry-leading performance for a wide variety of programming languages and platforms, including:

- C, C++, .NET[1], Java, and Ada[2] development platforms
- Windows, Linux, Solaris, Android, AIX, and other enterprise-class systems
- VxWorks, INTEGRITY, LynxOS, and other real-time and/or embedded systems

Applications written in different programming languages, running on different hardware under different operating systems, can interoperate seamlessly over *Connext DDS*, allowing disparate applications to work together in even very complex systems.

1.3.5 Interoperate with Databases, Event Engines, and JMS Systems

Connext DDS provides connections between its middleware core and many types of enterprise software. Simple-but-powerful integrations with databases, Complex Event Processing (CEP) engines, and other middlewares ensure that *Connext DDS* can bind together your real-time and enterprise systems.

For more information about interoperability with other middleware implementations, please consult your RTI account representative.

1.4 What Can Connext DDS Do?

Under the hood, *Connext DDS* goes beyond basic publish-subscribe communication to target the needs of applications with high-performance, real-time, and/or low-overhead requirements. It features:

- **Peer-to-peer, publish-subscribe communications**—The most elegant, flexible data communications model.

---

[1]The *Connext DDS* .NET language binding is currently supported for C# and C++/CLI.

[2]Ada support requires a separate add-on product, *Ada Language Support*.

- Simplified distributed application programming
- Time-critical data flow with minimal latency
- Clear semantics for managing multiple sources of the same data.
- Customizable Quality of Service and error notification.
- Guaranteed periodic messages, with minimum and maximum rates set by subscriptions
- Notifications when applications fail to meet their deadlines.
- Synchronous or asynchronous message delivery to give applications control over the degree of concurrency.
- Ability to send the same message to multiple subscribers efficiently, including support for reliable multicast with customizable levels of positive and negative message acknowledgement.

- **Request-Reply Communications**—As applications become more complex, it often becomes necessary to use other communication patterns in addition to publish-subscribe. Sometimes an application needs to get a one-time snapshot of information; for example, to make a query into a database or retrieve configuration parameters that never change. Other times an application needs to ask a remote application to perform an action on its behalf. To support these scenarios, *Connext DDS* includes support for the request-reply communication pattern. The *Requester* (service consumer or client) sends a request message and waits for a reply message. The *Replier* (service provider) receives the request message and responds with a reply message.
- **Reliable messaging**—Enables subscribing applications to customize the degree of reliability required. Reliability can be tuned; you can guarantee delivery no matter how many retries are needed or try messages only once for fast and deterministic performance. You can also specify any settings in between. No other middleware lets you make this critical trade off on a per-message stream basis.
- **Multiple communication networks**—Multiple independent communication networks (*DDS domains*), each using *Connext DDS,* can be used over the same physical network to isolate unrelated systems or subsystems. Individual applications can participate in one or multiple DDS domains.
  - Symmetric architecture—Makes your application robust. No central server or privileged nodes means your system is robust to application and/or node failures.
  - Dynamic—Topics, subscriptions, and publications can be added or removed from the system at any time.
  - Multiple network transports—*Connext DDS* includes support for UDP/IP (IPv4 and IPv6)—including, for example, Ethernet, wireless, and Infiniband networks—and shared memory transports. It also includes the ability to dynamically plug in additional network transports and route messages over them. It can be configured to operate over a variety of transport mechanisms, including backplanes, switched fabrics, and other networking technologies.

- Multi-platform and heterogeneous system support—Applications based on *Connext DDS* can communicate transparently with each other regardless of the underlying operating system or hardware. Consult the *Release Notes* to see which platforms are supported in this release.
- Vendor neutrality and standards compliance—The *Connext DDS* API complies with the DDS specification. On the network, it supports the open DDS Interoperability Protocol, Real-Time Publish Subscribe (RTPS), which is also an open standard from the OMG.

1.5 Am I Better Off Building My Own Middleware?

Sometimes application projects start with minimal networking needs. So it's natural to consider whether building a simplified middleware in-house is a better alternative to purchasing a more-complex commercial middleware. While doing a complete Return on Investment (ROI) analysis is outside the scope of this document, with *Connext DDS*, you get a rich set of *high-performance* networking features by just turning on configuration parameters, often without writing a single line of additional code.

RTI has decades of experience with hundreds of applications. This effort created an integrated and time-tested architecture that seamlessly provides the flexibility you need to succeed now and grow into the future. Many features require fundamental support in the core of the middleware and cannot just be cobbled onto an existing framework. It would take many man-years of effort to duplicate even a small subset of the functionality to the same level of stability and reliability as delivered by RTI.

For example, some of the middleware functionality that your application can get by just enabling configuration parameters include:

- Tuning reliable behavior for multicast, lossy and high-latency networks.
- Supporting high data-availability by enabling redundant data sources (for example, "keep receiving data from source A. If source A stops publishing, then start receiving data from source B"), and providing notifications when applications enter or leave the network.
- Optimizing network and system resources for transmission of periodic data by supporting time-based filtering of data (example: "receive a DDS sample no more than once per second") and deadlines (example: "expect to receive a DDS sample at least once per second").

Writing network code to connect a few nodes is deceptively easy. Making that code scale, handle all the error scenarios you will eventually face, work across platforms, keep current with technology, and adapt to unexpected future requirements is another matter entirely. The initial cost of a custom design may seem tempting, but beware; robust architectures take years to evolve. Going with a fire-tested, widely used design assures you the best performance and functionality that will scale with your distributed application as it matures. And it will minimize the profound cost of going down the wrong path.

Chapter 2 Navigating the Directories

To install *Connext DDS*, see the separate *RTI Connext DDS Installation Guide*.

After installing *Connext DDS*, your installation directory will include:

- **/bin** — Batch files for running the target package installer, utilities, code generator, etc.
- **/doc/manuals** - PDF documentation.
- **/doc/api** - API Reference HTML documentation.
- **/include** — Header files for C and C++ APIs, specification files for Ada
- **/lib** — Library files
- **/resource** — Document-format definitions, template files used by rtiddsgen, as well as the run-time components used by the RTI tools and services, including the RTI libraries and JRE.
- **/uninstall** — Uninstaller
- **/README.html** — Starting page for accessing the API Reference HTML documentation

The **doc** directory contains the *Connext DDS* library information in PDF and HTML formats. You may want to bookmark the **doc** directory since you will be referring to this page a lot as you explore the RTI technology platform.

Examples are also available, see 1.2 Paths Mentioned in Documentation on page 4.

See the instructions in each example's **README_ME.txt** file. These examples include:

- **hello_world**: This example demonstrates a simple "hello world" built with *Connext DDS*: it does nothing but publish and subscribe to short strings of text. This example is described in detail in 4.1 Building and Running “Hello, World” on page 14.

- **hello_builtin, hello_idl, hello_dynamic**[a]: These examples demonstrate some more of the unique capabilities of *Connext DDS*: strongly typed data, QoS-based customization of behavior, and industry-leading performance. These examples are described in Chapter 5 Capabilities and Performance on page 39 and Chapter 7 Design Patterns for High Performance on page 80.
- **hello_world_request_reply:** This example demonstrates how to use 4.3.7 Requesters and Repliers on page 37. The Replier is capable of computing the prime numbers below a certain positive integer; the Requester will request these numbers. The Replier provides the prime numbers as they are being calculated, sending multiple replies. See 4.2 Building and Running a Request-Reply Example on page 24.
- **news:** This example demonstrates a subset of the rich functionality *Connext DDS* offers, including flexible support for historical and durable data, built-in data caching, powerful filtering capabilities, and tools for working with periodic data. This example is described in Chapter 6 Design Patterns for Rapid Development on page 57.

You can find more examples at http://www.rti.com/examples. This page contains example code snippets on how to use individual features, examples illustrating specific use cases, as well as performance test examples.

Connext DDS supports the C, C++, C++/CLI, C#, Java, and Ada[b] programming languages. There are two versions of the C++ API, the traditional C++ API and the modern C++ PSM API. While we will examine the C++, Java, and Ada examples in the following sections, you can access similar code in the language of your choice.

---

[a]hello_dynamic is not provided for Ada.

[b]Ada support requires a separate component, *Ada Language Support*.

Chapter 3 Selecting a Development Environment

You can develop applications with *Connext DDS* either by building and executing commands from a shell window, or by using a development environment like Microsoft® Visual Studio®, Eclipse™, or GPS from AdaCore[1].

The following discusses building and running the provided examples from the command line or Visual Studio. Then we'll step through the details in 4.1 Building and Running “Hello, World” on page 14.

The **<path to examples>** is described in 1.2 Paths Mentioned in Documentation on page 4.

3.1 Using the Command-Line Tools

For most Java applications, you can use the following script files to build and execute.

- **<path to examples>/connext_dds/java/<*example*>/build.sh** (or **build.cmd** on Windows systems) —Builds Java source files; no parameters are needed.
- **<path to examples>/connext_dds/java/<*example*>/run.sh** (or **run.cmd** on Windows systems)
  —Runs the main program, Hello, in either Publisher or Subscriber mode. It accepts the following parameters:
  - [pub|sub] —Runs as publisher or subscriber
  - –verbose —Increases output verbosity

  (This script accepts other options, which we will discuss later.)

[1]Ada support requires a separate add-on product, *Ada Language Support.*

The **hello_world** example uses the *rtiddsgen* tool to generate code, instead of these scripts. The steps to build and run **hello_world** are described in 4.1 Building and Running "Hello, World" on page 14.

For C, C++, and Ada applications:

- You can use the **make** command with this makefile:

  <path to examples>/connext_dds/<*language*>/<*example*>/makefile_<*example*>_<*architecture*>

For Java users:

- The native libraries used by the RTI Java API require the Visual Studio redistributable libraries on the target machine. You can obtain this package from Microsoft or RTI.

3.2 Using Microsoft Visual Studio

For C, C++, and C# users: Please use a supported version of Microsoft Visual Studio to build and run the examples. Supported versions of Visual Studio are listed in the *Platform Notes*.

Connext DDS includes solutions and project files for Microsoft Visual Studio in **<*path to examples*>\connext_dds\[c|c++|cs]\<*example*>**.

To use these solution files:

1. Start Visual Studio.
2. Select **File, Open, Project/Solution**.
3. In the File dialog, select the solution file for your architecture. For example, a solution file for Visual Studio 2012 for 32-bit platforms is in **<*path to examples*>\connext_dds\[c|c++|cs]\<*example*>\win32\Hello-i86Win32VS2012.sln**.

Chapter 4 A Quick Overview

Before continuing, follow the instructions in the *RTI Connext DDS Installation Guide*.

This chapter gets you up and running with *Connext DDS*. First you will build and run your first *Connext DDS*-based application. Then we'll take a step back to learn about the general concepts in *Connext DDS* and show how they are applied in the example application you ran.

4.1 Building and Running "Hello, World"

Let's start by compiling and running Hello World, a basic program that publishes information over the network.

For now, do not worry about understanding the code (we start covering it in Capabilities and Performance (Chapter 5 on page 39). Use the following instructions to run your first middleware program using *Connext DDS*.

Find a new version of this exercise, in an updated getting started guide, at https://community.rti.com/documentation.

4.1.1 Step 1: Set Up the Environment

There are a few things to take care of before you start working with the example code.

4.1.1.1 Set Up the Environment on Your Development Machine

1. Set the **NDDSHOME** environment variable.

   Set the environment variable **NDDSHOME** to the *Connext DDS* install directory. (*Connext DDS* itself does not require that you set this environment variable. It is used in the scripts used to build and run the **examples** code because it is a simple way to locate the install directory. You may or may not choose to use the same mechanism when you create scripts to

build and/or run your own applications.)

The default installation paths are described in 1.2 Paths Mentioned in Documentation on page 4:

If you have multiple versions of Connext DDS installed:

Connext DDS does not require that you have the environment variable NDDSHOME set at run time. However, if it is set, *Connext DDS* will use it to load certain configuration files. Additionally, you may have previously set your path based on the value of that variable. Therefore, if you have NDDSHOME set, be sure it is pointing to the right copy of *Connext DDS*.

2. Update your path.

   Add *Connext DDS's* **bin** directory to your path. This will allow you to run some of the simple command-line utilities included in your distribution without typing the full path to the executable.

   On UNIX-based systems:

   Add the directory to your PATH environment variable.

   On Windows systems:

   Add the directory to your Path environment variable.

3. If you will be using the separate add-on product, *Ada Language Support*:

   Add **$NDDSHOME/lib/gnat** to your ADA_PROJECT_PATH environment variable. This directory contains Ada project files that will be used in the generated example project file.

   Make sure the Ada compiler, **gprbuild**, is in your path. The makefile used by the example assumes that **gprbuild** is in your path.

   On UNIX-based systems:

   Add the path to **gprbuild** to your PATH environment variable.

4. Make sure Java is available (only needed if you will be developing in Java).

   Ensure that appropriate **java** and **javac** executables are in your path. They can be found within the **bin** directory of your JDK installation. The *Release Notes* list the Java versions that are supported.

   On Linux systems:

   Note that GNU java (from the GNU Classpath project) is not supported—and will typically not work—but is in the path by default on many Linux systems.

5. Make sure the preprocessor is available.

   Check whether the C preprocessor (e.g., **cpp**) is in your search path. This step is optional, but makes code generation with the *rtiddsgen* utility more convenient.Capabilities and Performance (Chapter 5 on page 39) describes how to use *rtiddsgen*.

   On Windows systems, if Microsoft Visual Studio is installed:

   Running the script **vcvars32.bat**, **vsvars32.bat**, or **vcvarsall.bat** (depending on your version of Visual Studio) will update the path for a given command prompt. If the Visual Studio installer did not add **cl** to your path already, you can run this script before running *rtiddsgen*.

   On Windows systems, if Microsoft Visual Studio is not installed:

   This is often the case with Java users. You can either choose not to use a preprocessor or to obtain a no-cost version of Visual Studio from Microsoft's web site.

6. Get your project files ready.

   On Windows systems:

   If you installed *Connext DDS* in **C:\Program Files** or another system directory, Microsoft Visual Studio may present you with a warning message when you open a solution file from the installation directory. If you see this dialog, you may want to copy the example directory somewhere else, as described above.

   On UNIX-based systems:

   The makefiles that RTI provides with the example code are intended to be used with the GNU distribution of the make utility. On modern Linux systems, the make binary typically is GNU make.

On other systems, GNU make is called **gmake**. For the sake of clarity, the name **gmake** is used below. Make sure that the GNU make binary is on your path before continuing.

4.1.1.2 Set Up the Environment on Your Deployment Machine

Some configuration has to be done for the machine(s) on which you run your application; the RTI installer can't do that for you, because those machines may or may not be the same as where you created and built the application.

1. Make sure Java is available (only needed if you will be developing in Java).

   Ensure that appropriate **java** and **javac** executables are in your path. They can be found within the **bin** directory of your JDK installation. The *Release Notes* list the Java versions that are supported.

   On Linux systems: Note that GNU java (from the GNU Classpath project) is not supported—and will typically not work—but is in the path by default on many Linux systems.

2. Make sure the dynamic libraries are available.

   Make sure that your application can load the *Connext DDS* dynamic libraries. If you use C, C++, or Ada with *static* libraries (the default configuration in the examples covered in this document), you can skip this step. However, if you plan to use *dynamic* libraries, or Java or .NET[a] (which always use dynamic libraries), you will need to modify your environment as described here.

   To see if dynamic libraries are supported for your machine's architecture, see the *Connext DDS Core Libraries Platform Notes*[b].For more information about where the Windows OS looks for dynamic libraries, see: http://msdn.microsoft.com/en-us/library/ms682586.aspx.

   C/C++:

   The dynamic libraries needed by C or C++ applications are in the directory **${NDDSHOME}/lib/<architecture>**. The dynamic libraries needed by Ada applications are in the directory **${NDDSHOME}/lib/GNATgcc/relocatable**.

   On UNIX-based systems: Add this directory to your LD_LIBRARY_PATH environment variable.

   On macOS systems: Add this directory to your DYLD_LIBRARY_PATH environment variable.

---

[a]*Connext DDS* .NET language binding is currently supported for C# and C++/CLI.

[b]In the *Platform Notes*, see the "Building Instructions..." table for your target architecture.

On Windows systems: Add this directory to your Path environment variable or copy the DLLs into the directory containing your executable.

On AIX systems: Add this directory to your LIBPATH environment variable.

Java:

The native dynamic libraries needed by Java applications are in the directory **${NDDSHOME}/lib/<architecture>**. The native dynamic libraries needed by Ada applications are in the directory **$NDDSHOME/lib/<architecture>**.

- On UNIX-based systems: Add this directory to your LD_LIBRARY_PATH environment variable.
- On macOS systems: Add this directory to your DYLD_LIBRARY_PATH environment variable.
- On Windows systems: Add this directory to your Path environment variable.
- On AIX systems: Add this directory to your LIBPATH environment variable.

Java .jar files are in the directory **${NDDSHOME}/lib/java**. They will need to be on your application's class path.

.NET:

On Windows systems: The dynamic libraries needed by .NET applications are in the directory **%NDDSHOME%\lib\i86Win32VSnnnn**, where *nnnn* represents the Visual Studio version number. You will need to either copy the DLLs from that directory to the directory containing your executable, or add the directory containing the DLLs to your Path environment variable[a]. (If the .NET framework is unable to load the dynamic libraries at run time, it will throw a System.IO.FileNotFoundException and your application may fail to launch.)

4.1.2 Step 2: Compile the Hello World Program

The same example code is provided in C, Traditional C++, C#, Java, and Ada[b]. The following instructions cover C++, Java, and Ada in detail; the procedures for C and C# are very similar. The same source code can be built and run on different architectures. Examples for the Modern C++ API are provided in the RTI Community portal.

---

[a]The file nddsdotnet.dll (or nddsdotnetd.dll for debug) must be in the executable directory. Visual Studio will, by default, do this automatically.

[b]Ada support requires a separate add-on product, Ada Language Support.

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

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

Default path to the examples:

- macOS systems: **/Users/<*your user name*>/rti_workspace/6.0.1/examples**
- Linux systems: **/home/<*your user name*>/rti_workspace/6.0.1/examples**
- Windows systems: ***<your Windows documents folder>*\rti_workspace\6.0.1\examples**

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

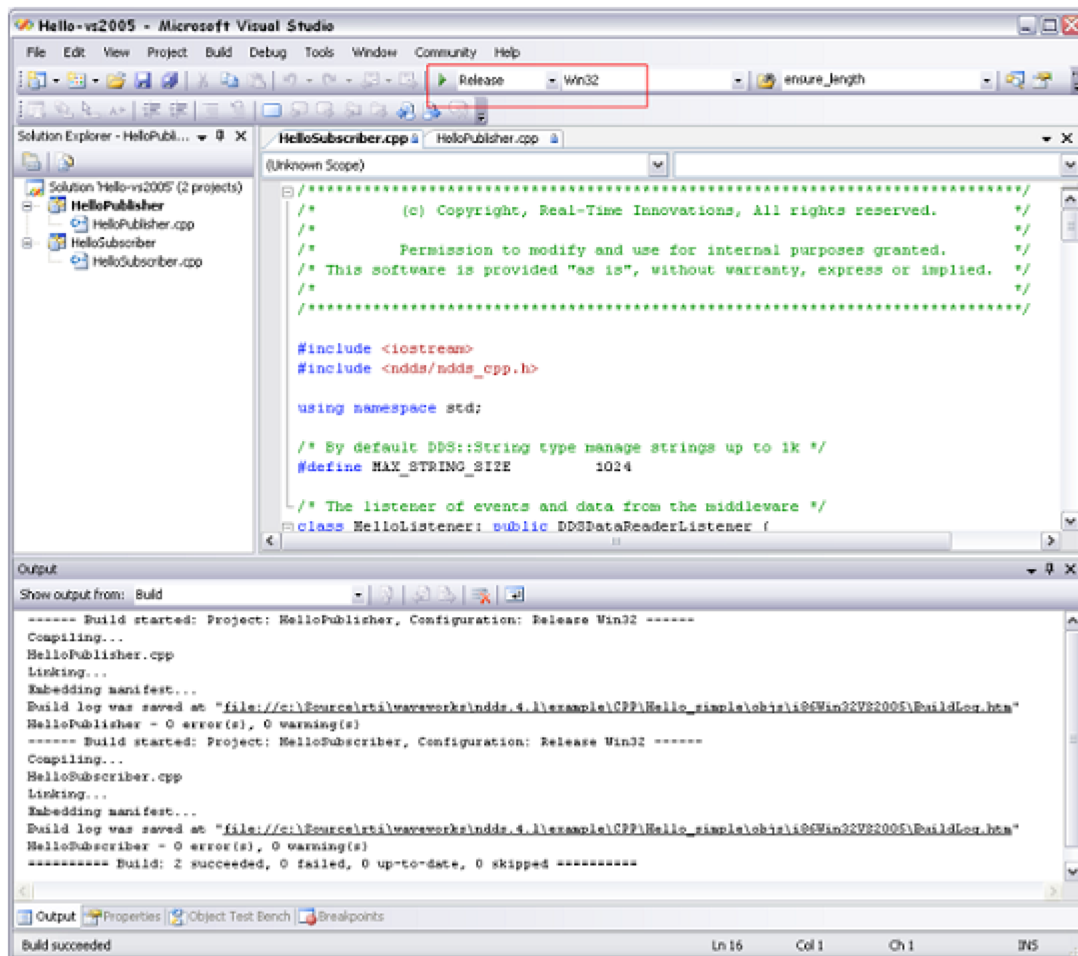
Note: You can specify a different location for **rti_workspace**. You can also specify that you do not want the examples copied to the workspace. For details, see *Controlling Location for RTI Workspace and Copying of Examples* in the *RTI Connext DDS Installation Guide*.

The instructions also focus on Windows and UNIX-based systems. If you will be using an embedded platform, see the *Embedded Systems Addendum* for more instructions.

C++ on Windows Systems:

1. In the Windows Explorer, go to ***<path to examples>*\connext_dds\c++\hello_world\win32** and open the Visual Studio solution file for your architecture. For example, the file for Visual Studio 2013 for 32-bit platforms is **HelloWorld-i86Win32VS2013.sln**.

   Note: If your Windows SDK Version is not 10.0.15063.0, you may be prompted to retarget the file. If this happens, in the Retarget Projects window that appears, select an installed version of Windows SDK and click OK.
2. The Solution Configuration combo box in the toolbar indicates whether you are building debug or release executables; select **Release**. Select **Build Solution** from the **Build** menu.



C++ on UNIX-based Systems:

From your command shell, go to **<path to examples>/connext_dds/c++/hello_world/** and type:

```
> gmake -f make/makefile_HelloWorld_<architecture>
```

where <*architecture*> is one of the supported architectures; see the contents of the **examples** directory for a list of available architectures. (If you do not see a makefile for your architecture, please refer to 5.3.3 Generating Code with RTI Code Generator on page 49 to learn how to generate a makefile or project files for your platform). This command will build a release executable.

To build a debug version instead:

```
> gmake -f make/makefile_HelloWorld_<architecture> DEBUG=1
```

Java on Windows systems:

From your command shell, go to **<path to examples>\connext_dds\java\hello_world** and type:

```
> build
```

Java on UNIX-based systems:

From your command shell, go to **<path to examples>/connext_dds/java/hello_world** and type:

```
> ./build.sh
```

ADA on UNIX-based systems:

From your command shell, go to **<path to examples>/connext_dds/ada/hello_world/** and type:

```
> gmake -f make/makefile_HelloWorld_<architecture>
```

where <*architecture*> is one of the supported architectures; see the contents of the **examples** directory for a list of available architectures. (If you do not see a makefile for your architecture, please refer to 5.3.3 Generating Code with RTI Code Generator on page 49 to learn how to generate a makefile or project files for your platform). This command will build a release executable.

To build a debug version instead:

```
> gmake -f make/makefile_HelloWorld_<architecture> DEBUG=1
```

4.1.3 Step 3: Start the Subscriber

C++ on Windows systems:

From your command shell, go to **examples\connext_dds\c++\hello_world** and type:

```
> objs\<architecture>\HelloWorld_subscriber.exe
```

where <architecture> is one of the supported architectures; see the contents of the **examples** directory for a list of available architectures. For example, the Windows architecture name corresponding to 32-bit Visual Studio 2013 is i86Win32VS2013.

C++ on UNIX-based systems:

From your command shell, go to **/examples/connext_dds/c++/hello_world** and type:

```
> objs/<architecture>/HelloWorld_subscriber
```

where <*architecture*> is one of the supported architectures; see the contents of the **examples** directory for a list of available architectures. For example, a Red Hat® Enterprise Linux architecture is i86Linux3gcc-c4.8.2.

Java:

(As described in 4.1.1 Step 1: Set Up the Environment on page 14, you should have already set your path appropriately so that the example application can load the native libraries on which *Connext DDS* depends. If you have not, you can set the variable RTI_EXAMPLE_ARCH in your command shell—e.g., to i86Win32VS2013 or i86Linux3gcc4.8.2—and the example launch scripts will use it to set your path for you.)

Java on Windows systems:

From your command shell, go to **examples\connext_dds\java\hello_world** and type:

```
> runSub
```

Java on UNIX-based systems:

From your command shell, go to **examples/connext_dds/java/hello_world** and type:

```
> ./runSub.sh
```

Ada on UNIX-based systems:

From your command shell, go to **/examples/connext_dds/ada/hello_world** and type:

```
> objs/<architecture>/hellosubscriber
```

Where <architecture> is one of the supported architectures; see the content of the **examples** directory for a list of available architectures. For example, an architecture name corresponding to a Red Hat Enterprise Linux system is x64Linux2.6gcc4.4.5.

4.1.4 Step 4: Start the Publisher

Connext DDS interoperates across all of the programming languages it supports, so you can choose whether to run the publisher in the same language you chose for the subscriber or a different language.

C++ on Windows systems:

From a different command shell, go to **examples\connext_dds\c++\hello_world** and type:

```
> objs\<architecture>\HelloWorld_publisher.exe
```

where <*architecture*> is one of the supported architectures; see the contents of the **examples** directory for a list of available architectures. For example, the Windows architecture name corresponding to 32-bit Visual Studio 2013 is i86Win32VS2013.

C++ on UNIX-based systems:

From a different command shell, go to **examples/connext_dds/c++/hello_world** and type:

```
> objs/<architecture>/HelloWorld_publisher
```

where <*architecture*> is one of the supported architectures; see the contents of the **example** directory for a list of available architectures. For example, a Red Hat Enterprise Linux architecture name is i86Linux3gcc4.8.2.

Java:

(As described above, you should have already set your path appropriately so that the example application can load the native libraries on which *Connext DDS* depends. If you have not, you can set the variable RTI_EXAMPLE_ARCH in your command shell—e.g., to i86Win32VS2013 or i86Linux3gcc4.8.2.—

and the example launch scripts will use it to set your path for you.)

Java on Windows systems:

From a different command shell, go to **<path to examples>\connext_dds\java\hello_world** and type:

```
> runPub
```

Java on UNIX-based systems:

From a different command shell, go to **<path to examples>/connext_dds/java/hello_world** and type:

```
> ./runPub.sh
```

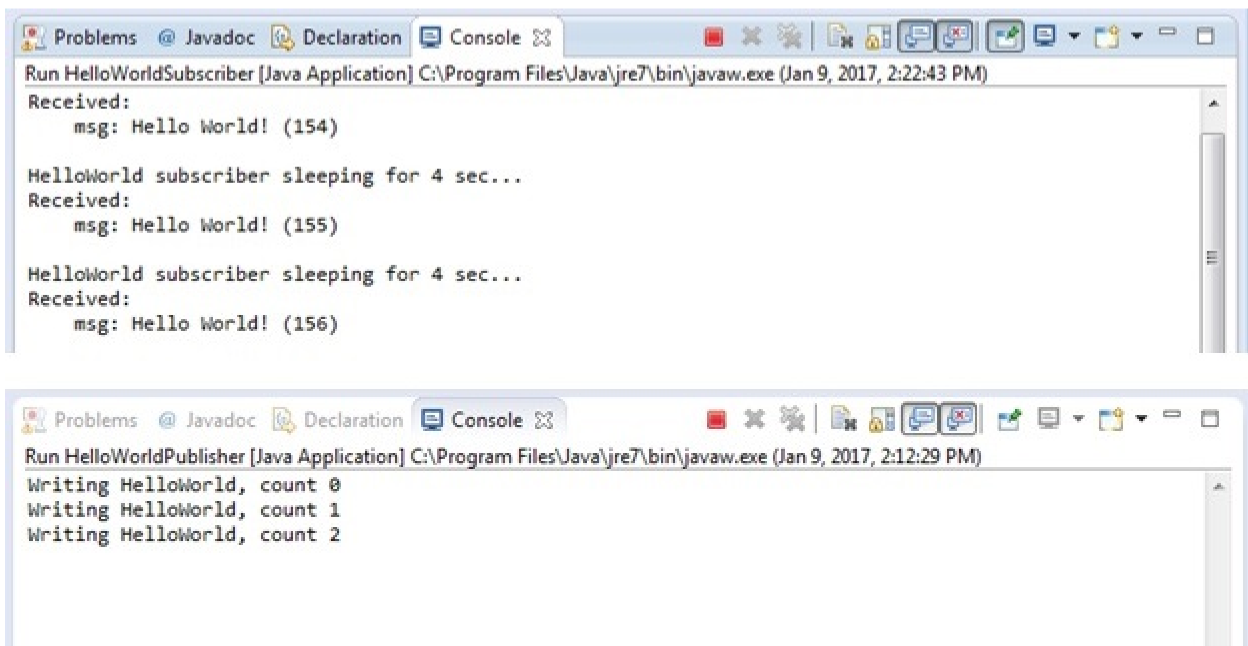
Ada on UNIX-based systems:

From a different command shell, go to **<path to examples>/connext_dds/ada/hello_world** and type:

```
> objs/<architecture>/hellopublisher
```

Where <*architecture*> is one of the supported architectures; see the contents of the **example** directory for a list of available architectures. For example, an architecture name corresponding to a Red Hat Enterprise Linux system is i86Linux3gcc4.8.2.

When you run the publishing and subscribing applications, you should see output similar to the following:



Congratulations! You've run your first *Connext DDS* program!

4.2 Building and Running a Request-Reply Example

This section describes the Request-Reply communication pattern, which is only available with the RTI Connext DDS Professional, Secure, Basic, and Evaluation package types.

The section on 4.3.7 Requesters and Repliers on page 37 contains an introduction to the *Connext DDS* Request-Reply API. More detailed information is available in the *RTI Connext DDS Core Libraries User's Manual* (see *Part 4: Request-Reply Communication)* and in the API Reference HTML documentation (open **<NDDSHOME>/README.html** and select a programming language; then select **Modules, RTI Connext DDS API Reference, RTI Connext Messaging API Reference**). See 1.1 A Guide to the Provided Documentation on page 2.

Connext DDS provides the libraries that you will need when compiling an application that uses the Request-Reply API.

- In C, you need the additional **rticonnextmsgc** libraries and you will use a set of macros that instantiate DDS type-specific code. You will see how to do this in the code example.
- In C++, you need the additional **rticonnextmsgcpp** libraries and the header file **ndds/ndds_requestreply_cpp.h**.
- In Java, you need an additional JAR file: **rticonnextmsg.jar**.
- In .NET (C# and C++/CLI), you need an additional assembly:
  - For .NET 2.0: **rticonnextmsgdotnet.dll**
  - For .NET 4.0: **rticonnextmsgdotnet40.dll**
  - For .NET 4.5: **rticonnextmsgdotnet45.dll**
  - For .NET 4.5.1: **rticonnextmsgdotnet451.dll**
  - For .NET 4.6: **rticonnextmsgdotnet46.dll**

To set up your environment follow the same instructions in 4.1.1 Step 1: Set Up the Environment on page 14.

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

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

Default path to the examples:

- macOS systems: **/Users/*<your user name>*/rti_workspace/6.0.1/examples**
- Linux systems: **/home/*<your user name>*/rti_workspace/6.0.1/examples**

- Windows systems: **<*your Windows documents folder*>\rti_workspace\6.0.1\examples**

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

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

The Request-Reply examples are:

- **examples/connext_dds/c/hello_world_request_reply**
- **examples/connext_dds/c++/ hello_world_request_reply**
- **examples/connext_dds/cs/hello_world_request_reply**
- **examples/connext_dds/java/hello_world_request_reply**

To compile the examples, follow the instructions in 4.1.2 Step 2: Compile the Hello World Program on page 18. Similar makefiles for UNIX-based systems and scripts for Java and Visual Studio projects are provided in these examples. Instructions for running the examples are in **READ_ME.txt** in the example directories. See the instructions in each example's **README_ME.txt** file.

4.3 An Introduction to DDS

Connext DDS is a software connectivity framework for real-time distributed applications. It provides the middleware communications service that programmers need to distribute time-critical data between embedded and/or enterprise devices or nodes. *Connext DDS* uses a *publish-subscribe* communications model to make data-distribution efficient and robust. *Connext DDS* also supports the Request-Reply communication pattern[a].

Connext DDS implements the *Data-Centric Publish-Subscribe (DCPS)* API of the OMG's specification, *Data Distribution Service (DDS) for Real-Time Systems.* DDS is the first standard developed for the needs of real-time systems, providing an efficient way to transfer data in a distributed system. With *Connext DDS*, you begin your development with a fault-tolerant and flexible communications infrastructure that will work over a wide variety of computer hardware, operating systems, languages, and networking transport protocols. *Connext DDS* is highly configurable, so programmers can adapt it to meet an application's specific communication requirements.

---

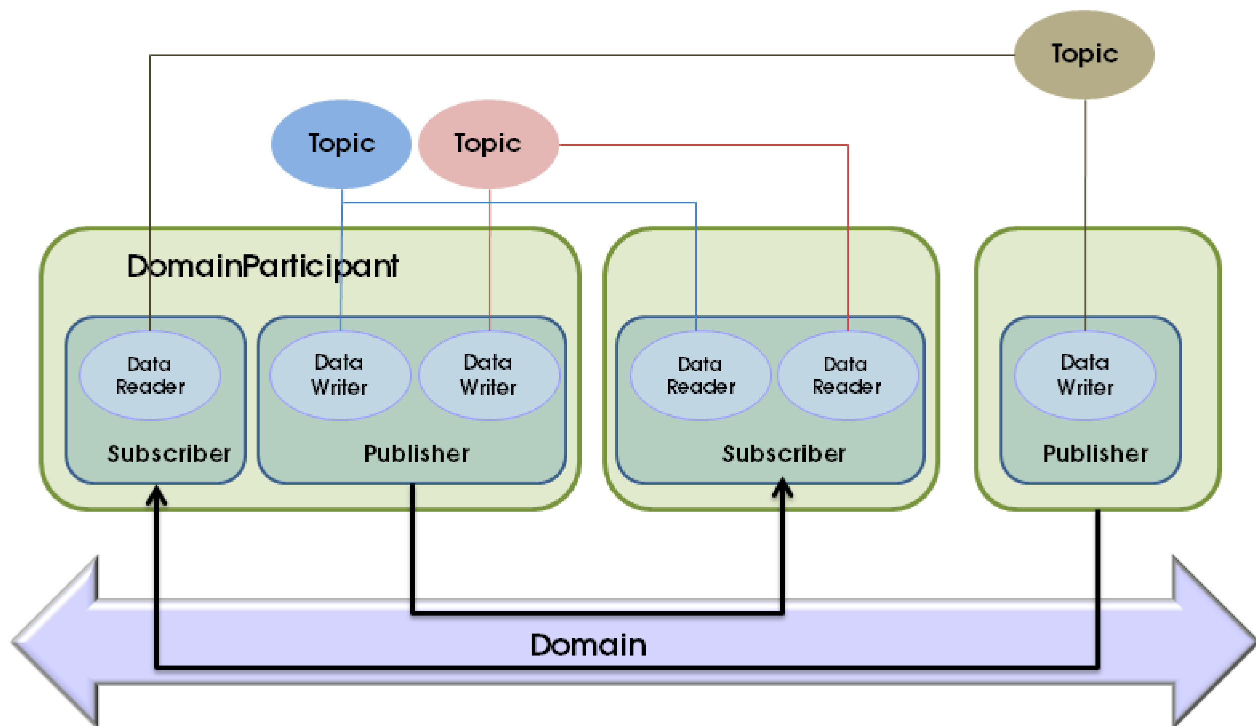
[a]Request-reply communication is available only when using the *Connext DDS* Professional, Secure, Basic, and Evaluation package types.

4.3.1 An Overview of DDS Objects

The primary objects in DDS are:

- 4.3.2 DomainParticipants below
- 4.3.3 Publishers and DataWriters on page 29
- 4.3.4 Subscribers and DataReaders on page 31
- 4.3.5 Topics on page 35
- 4.3.6 Keys and DDS Samples on page 36
- 4.3.7 Requesters and Repliers on page 37

Figure 4.1: *Connext DDS* Components



4.3.2 DomainParticipants

A *DDS domain* is a concept used to bind individual applications together for communication. To communicate with each other, *DataWriters* and *DataReaders* must have the same *Topic* of the same DDS data type and be members of the same DDS domain. Each DDS domain has a unique integer domain ID.

Applications in one DDS domain cannot subscribe to data published in a different DDS domain. Multiple DDS domains allow you to have multiple virtual distributed systems on the same physical network. This

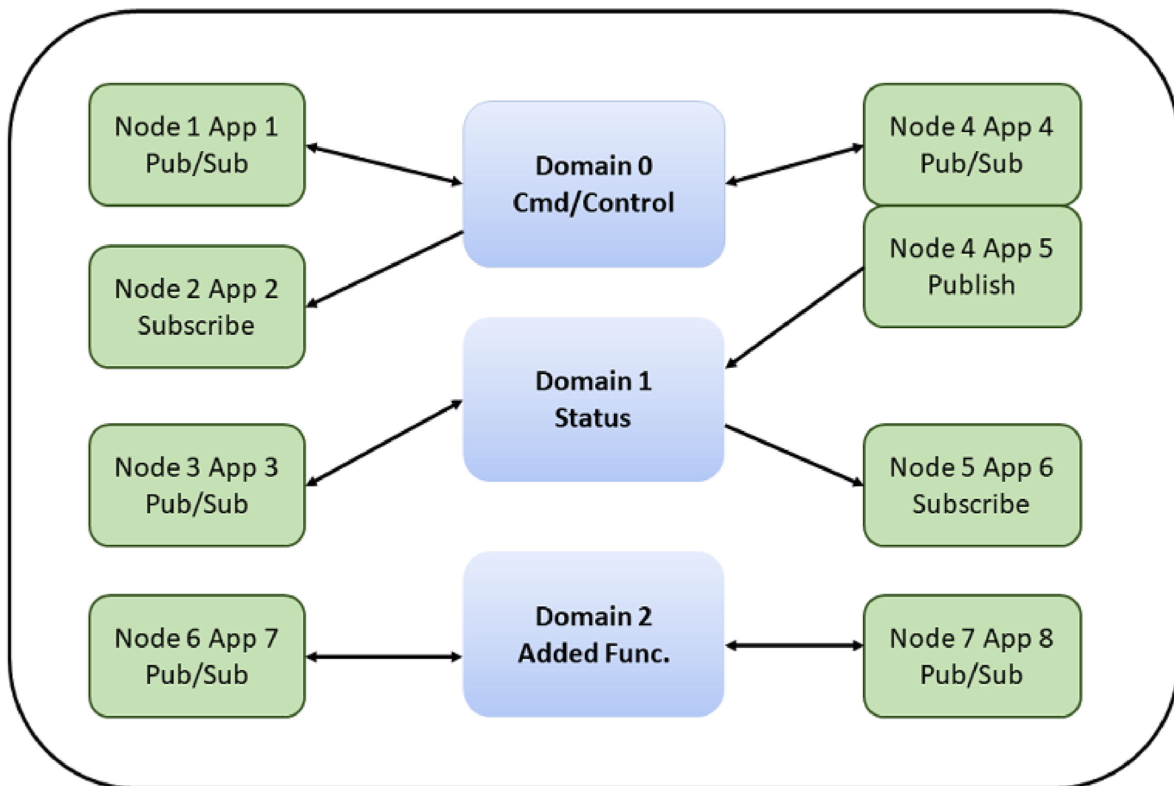
can be very useful if you want to run multiple independent tests of the same applications. You can run them at the same time on the same network as long as each test runs in a different DDS domain. Another typical configuration is to isolate your development team from your test and production teams: you can assign each team or even each developer a unique DDS domain.

DomainParticipant objects enable an application to exchange messages within DDS domains. An application must have a *DomainParticipant* for every DDS domain in which the application will communicate. (Unless your application is a bridging application, it will typically participate in only one DDS domain and have only one *DomainParticipant*.)

A *DomainParticipant* is analogous to a JMS Connection.

DomainParticipants are used to create *Topics, Publishers, DataWriters, Subscribers,* and *DataReaders* in the corresponding DDS domain.

Figure 4.2: Segregating Applications with Domains



The following code shows how to instantiate a *DomainParticipant*. You can find more information about all of the APIs in the API Reference HTML documentation.

To create a *DomainParticipant:*

In Traditional C++:

```
participant =
    DDSDomainParticipantFactory::get_instance()->create_participant(
      0, /* Domain ID */
      DDS_PARTICIPANT_QOS_DEFAULT, /* QoS */
      NULL, /* Listener */
      DDS_STATUS_MASK_NONE);
```

In Modern C++:

```
dds::domain::DomainParticipant participant(0);
```

In Java:

```
participant =
    DomainParticipantFactory.get_instance().create_participant(
      0, // Domain ID
      DomainParticipantFactory.PARTICIPANT_QOS_DEFAULT,
      null, // Listener
      StatusKind.STATUS_MASK_NONE);
```

In Ada[a]:

```
participant :=
    DDS.DomainParticipantFactory.Get_Instance.Create_Participant(
     0, --  Domain ID
     DDS.DomainParticipantFactory.PARTICIPANT_QOS_DEFAULT, -- QoS
     null, --  Listener
     DDS.STATUS_MASK_NONE);
```

As you can see, there are four pieces of information you supply when creating a new *DomainParticipant*:

- The ID of the DDS domain to which it belongs.
- Its qualities of service (QoS). The discussion on 4.3.2.1 What is QoS? on the next page gives a brief introduction to the concept of QoS; you will learn more in Chapter 5 Capabilities and Performance on page 39.
- Its listener and listener mask, which indicate the events that will generate callbacks to the *DomainParticipant*. You will see a brief example of a listener callback when we discuss *DataReaders* below. For a more comprehensive discussion of the *Connext DDS* status and notification system, see Chapter 4 in the *RTI Connext DDS Core Libraries User's Manual*.

(In the Modern C++ API the QoS and listener and mask are optional parameters.)

[a]Ada support requires a separate add-on product, ***Ada Language Support.***

4.3.2.1 What is QoS?

Fine control over Quality of Service (QoS) is perhaps the most important feature of *Connext DDS*. Each data producer-consumer pair can establish independent quality of service agreements—even in many-to-many topologies. This allows applications to support extremely complex and flexible data-flow requirements.

QoS policies control virtually every aspect of *Connext DDS* and the underlying communications mechanisms. Many QoS policies are implemented as "contracts" between data producers (*DataWriters*) and consumers (*DataReaders*); producers offer and consumers request levels of service. *Connext DDS* is responsible for determining if the offer can satisfy the request, thereby establishing the communication or indicating an incompatibility error. Ensuring that participants meet the level-of-service contracts guarantees predictable operation. For example:

Periodic producers can indicate the speed at which they can publish by offering guaranteed update deadlines. By setting a deadline, a producer promises to send updates at a minimum rate. Consumers may then request data at that or any slower rate. If a consumer requests a higher data rate than the producer offers, *Connext DDS* will flag that pair as incompatible and notify both the publishing and subscribing applications.

Producers may offer different levels of reliability, characterized in part by the number of past DDS data samples they store for retransmission. Consumers may then request differing levels of reliable delivery, ranging from fast-but-unreliable "best effort" to highly reliable in-order delivery. This provides per-data-stream reliability control. A single producer can support consumers with different levels of reliability simultaneously.

Other QoS policies control when *Connext DDS* detects nodes that have failed, set delivery order, attach user data, prioritize messages, set resource utilization limits, partition the system into namespaces, control durability (for fault tolerance) and much more. The *Connext DDS* QoS policies offer unprecedented flexible communications control. The *RTI Connext DDS Core Libraries User's Manual* contains details about all available QoS policies.

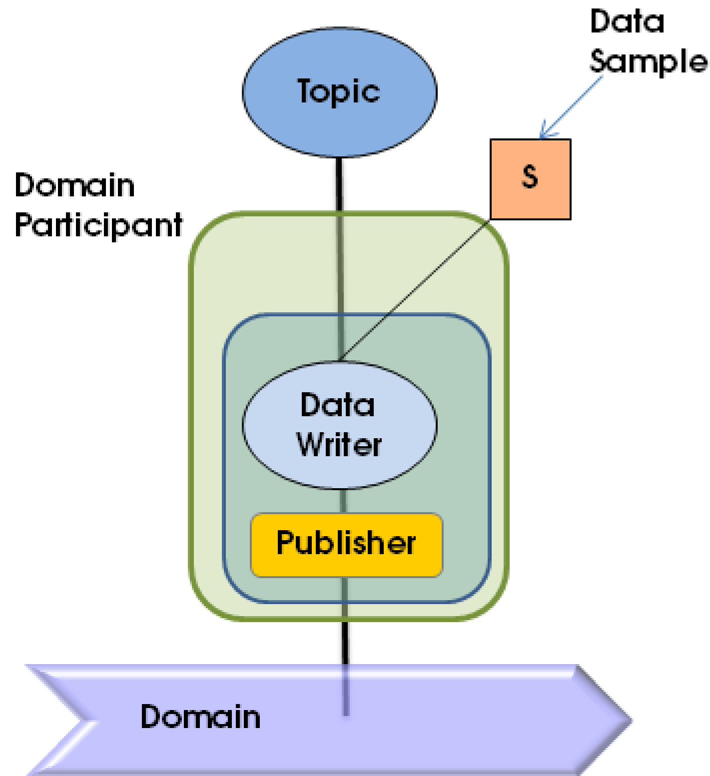
4.3.3 Publishers and DataWriters

An application uses a *DataWriter* to publish data into a DDS domain. Once a *DataWriter* is created and configured with the correct QoS settings, an application only needs to use the *DataWriter's* “write” operation to publish data.

A *DataWriter* is analogous to a JMS *TopicPublisher*. A *Publisher* is analogous to the producing aspect of a JMS *TopicSession*.

A *Publisher* is used to group individual *DataWriters*. You can specify default QoS behavior for a *Publisher* and have it apply to all the *DataWriters* in that *Publisher's* group.

Figure 4.3: Entities Associated with Publications



To create a *DataWriter:*

In Traditional C++:

```
data_writer = participant->create_datawriter(
    topic,
    DDS_DATAWRITER_QOS_DEFAULT, /* QoS */
    NULL, /* Listener */
     DDS_STATUS_MASK_NONE);
```

In Modern C++:

```
dds::pub::DataWriter<dds::core::StringTopicType> writer(
    rti::pub::implicit_publisher(participant), topic);
```

In Ada:

```
data_writer := participant.Create_DataWriter(
    topic,
    DDS.Publisher.DATAWRITER_QOS_DEFAULT, --  QoS
    null, --  Listener
    DDS.STATUS_MASK_NONE);
```

As you can see, each *DataWriter* is tied to a single topic. All data published by that *DataWriter* will be directed to that *Topic*.

As you will learn in Chapter 5 Capabilities and Performance on page 39, each *Topic*—and therefore all *DataWriters* for that *Topic*—is associated with a particular concrete DDS data type. The **write** operation, which publishes data, is type safe, which means that before you can write data, you must perform a type cast, like this (in Modern C++ the *DataWriter* is already typed and there is no cast to perform):

In Traditional C++:

```
string_writer = DDSStringDataWriter::narrow(data_writer);
```

In Java:

```
StringDataWriter dataWriter =
    (StringDataWriter) participant.create_datawriter(
        topic,
        Publisher.DATAWRITER_QOS_DEFAULT,
        null, // listener
        StatusKind.STATUS_MASK_NONE);
```

In Ada:

```
string_writer := DDS.Builtin_String_DataWriter.Narrow(data_writer);
```

Note that in this particular code example, you will not find any reference to the *Publisher* class. In fact, creating the *Publisher* object explicitly is optional, because many applications do not have the need to customize any behavior at that level. If you choose not to create a *Publisher*, *Connext DDS* will implicitly choose an internal *Publisher* object. If you *do* want to create a *Publisher* explicitly, create it with a call to **participant.create_publisher()** (you can find more about this method in the API Reference HTML documentation) and then simply replace the call to **participant.create_datawriter()** with a call to **publisher.create_datawriter()**.

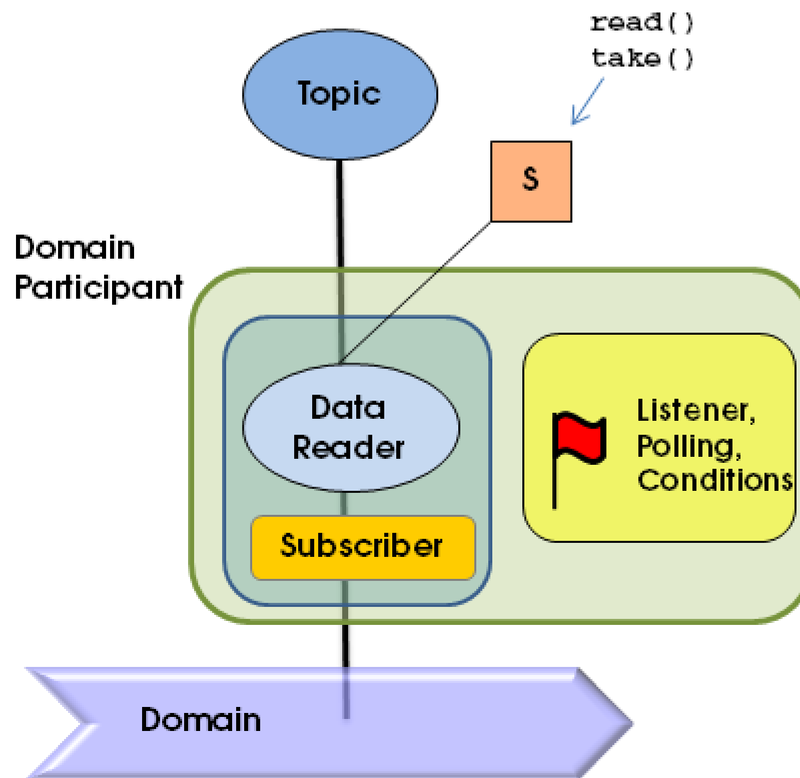
4.3.4 Subscribers and DataReaders

A *DataReader* is the point through which a subscribing application accesses the data that it has received over the network.

A *DataReader* is analogous to a JMS TopicSubscriber. A *Subscriber* is analogous to the consuming aspect of a JMS TopicSession.

Just as *Publishers* are used to group *DataWriters*, *Subscribers* are used to group *DataReaders*. Again, this allows you to configure a default set of QoS parameters and event handling routines that will apply to all *DataReaders* in the *Subscriber's* group.

Figure 4.4: Entities Associated with Subscriptions



Each *DataReader* is tied to a single topic. A *DataReader* will only receive data that was published on its *Topic*.

To create a *DataReader*:

In Traditional C++:

```
data_reader = participant->create_datareader(
    topic,
    DDS_DATAREADER_QOS_DEFAULT, /* QoS */
    &listener, /* Listener */
    DDS_DATA_AVAILABLE_STATUS);
```

In Modern C++:

```
// Without a listener:
dds::sub::DataReader<dds::core::StringTopicType> data_reader(
    rti::sub::implicit_subscriber(participant), topic);

// With a listener
dds::sub::DataReader<dds::core::StringTopicType> data_reader(
    rti::sub::implicit_subscriber(participant),
    topic,
    participant->default_datareader_qos(),
    &listener,
    dds::core::status::StatusMask::data_available());
```

In Java:

```
StringDataReader dataReader =
    (StringDataReader) participant.create_datareader(
        topic,
        Subscriber.DATAREADER_QOS_DEFAULT, // QoS
        new HelloSubscriber(), // Listener
        StatusKind.DATA_AVAILABLE_STATUS);
```

In Ada:

```
dataReader := DDS. Builtin_String_DataReader.Narrow(
    participant.Create_DataReader(
        topic.As_TopicDescription,
        DDS.Subscriber.DATAREADER_QOS_DEFAULT, --  QoS
        readerListener'Unchecked_Access, --  Listener
        DDS.DATA_AVAILABLE_STATUS));
```

Connext DDS provides multiple ways for you to access your data: you can receive it asynchronously in a listener, you can block your own thread waiting for it to arrive using a helper object called a *WaitSet*, or you can poll in a non-blocking fashion. This example uses the former mechanism, and you will see that it passes a non-NULL listener to the **create_datareader()** method. The listener mask (DATA_AVAILABLE_STATUS) indicates that the application is only interested in receiving notifications of newly arriving data.

Let's look at the callback implementation in the following example code.

In Traditional C++:

```
retcode = string_reader->take_next_sample(ptr_sample,info);
if (retcode == DDS_RETCODE_NO_DATA) {
   /* No more samples */
   break;

} else if (retcode != DDS_RETCODE_OK) {
   cerr << "Unable to take data from data reader, error "
   << retcode << endl;
   return;
}
```

In Modern C++:

```
dds::sub::LoanedSamples<dds::core::StringTopicType> samples = reader.take();
if (samples.length() == 0) {
    std::cout << "No samples\n"
}
// In case of error an exception is thrown
```

In Ada:

```
begin
    data_reader.Read_Next_Sample (ptr_sample, sample_info'Access);
   if sample_info.Valid_Data then
    Ada.Text_IO.Put_Line (DDS.To_Standard_String (ptr_sample));
   end if;
   exception
       when DDS.NO_DATA =>
        -- No more samples
       exit;
       when others =>
        Self.receiving := FALSE;
        exit;
   end;
```

The **take_next_sample()** method retrieves a single DDS data sample (i.e., a message) from the *DataReader*, one at a time without blocking. If it was able to retrieve a DDS sample, it will return **DDS_RETCODE_OK**. If there was no data to take, it will return **DDS_RETCODE_NO_DATA**. Finally, if it tried to take data but failed to do so because it encountered a problem, it will return **DDS_RETCODE_ERROR** or another **DDS_ReturnCode_t** value (see the API Reference HTML documentation for a full list of error codes).

Connext DDS can publish not only actual data to a *Topic*, but also *meta-data* indicating, for example, that an object about which the *DataReader* has been receiving data no longer exists. In the latter case, the info argument to **take_next_sample()** will have its **valid_data** flag set to false.

This simple example is interested only in DDS data samples, not meta-data, so it only processes “valid” data.

In Modern C++ we are retrieving all the samples at once.

In Traditional C++:

```
if (info.valid_data) {
    // Valid (this isn't just a lifecycle sample): print it
    cout << ptr_sample << endl;
}
```

In Modern C++:

```
for (auto sample : samples) {
    if (sample.info().valid()) {
        std::cout << sample.data() << std::endl;
    }
}
```

In Java:

```
try {

    String sample = stringReader.take_next_sample(info);
    if (info.valid_data) {
        System.out.println(sample);
    }
} catch (RETCODE_NO_DATA noData) {
    // No more data to read
    break;
} catch (RETCODE_ERROR e) {
    // An error occurred
    e.printStackTrace();
}
```

In Ada:

```
if sample_info.Valid_Data then
       --  Valid, print  it
       Ada.Text_IO.Put_Line (DDS.To_Standard_String (ptr_sample));
end if;
```

Note that in this particular code example, you will not find any reference to the *Subscriber* class. In fact, as with *Publishers*, creating the *Subscriber* object explicitly is optional, because many applications do not have the need to customize any behavior at that level. If you, like this example, choose not to create a *Subscriber*, the middleware will implicitly choose an internal *Subscriber* object. If you *do* want to create a *Subscriber* explicitly, create it with a call to **participant.create_subscriber** (you can find more about this method in the API Reference HTML documentation) and then simply replace the call to **participant.create_datareader** with a call to **subscriber.create_datareader**.

4.3.5 Topics

Topics provide the basic connection points between *DataWriters* and *DataReaders*. To communicate, the *Topic* of a *DataWriter* on one node must match the *Topic* of a *DataReader* on any other node.

A *Topic* is comprised of a *name* and a *DDS type*. The *name* is a string that uniquely identifies the *Topic* within a DDS domain. The *DDS type* is the structural definition of the data contained within the *Topic*; this capability is described in Chapter 5 Capabilities and Performance on page 39.

You can create a *Topic* with the following code:

In Traditional C++:

```
topic = participant->create_topic(
    "Hello, World", /* Topic name*/
    DDSStringTypeSupport::get_type_name(), /* Type name */
    DDS_TOPIC_QOS_DEFAULT, /* Topic QoS */
    NULL, /* Listener  */
    DDS_STATUS_MASK_NONE);
```

In Modern C++:

```
dds::topic::Topic<dds::core::StringTopicType> topic(
    participant, "Hello, World");
```

In Java:

```
Topic topic = participant.create_topic(
    "Hello, World", // Topic name
     StringTypeSupport.get_type_name(), // Type name
     DomainParticipant.TOPIC_QOS_DEFAULT, // QoS

     null,  // Listener
     StatusKind.STATUS_MASK_NONE);
```

In Ada:

```
topic := participant.Create_Topic(
    DDS.To_DDS_Builtin_String ("Hello, World"), --  Topic name
    DDS.Builtin_String_TypeSupport.Get_Type_Name, --  Type name
    DDS.DomainParticipant.TOPIC_QOS_DEFAULT, --  Topic QoS
    null, --  Listener
    DDS.STATUS_MASK_NONE);
```

Besides the new *Topic*'s name and DDS type, an application specifies three things:

- **A suggested set of QoS** for *DataReaders* and *DataWriters* for this *Topic*.
- **A listener and listener mask** that indicate which events the application wishes to be notified of, if any.

In this case, the *Topic's* DDS type is a simple string, a type that is built into the middleware.

4.3.6 Keys and DDS Samples

The data values associated with a *Topic* can change over time. The different values of the *Topic* passed between applications are called *DDS samples*. A *DDS sample* is analogous to a message in other publish-subscribe middleware.

An application may use a single *Topic* to carry data about many objects. For example, a stock-trading application may have a single topic, "Stock Price," that it uses to communicate information about Apple, Google, Microsoft, and many other companies. Similarly, a radar track management application may have a single topic, "Aircraft Position," that carries data about many different airplanes and other vehicles. These objects within a *Topic* are called instances. For a specific DDS data type, you can select one or more fields within the DDS data type to form a *key*. A *key* is used to uniquely identify one instance of a *Topic* from another instance of the same *Topic*, very much like how the primary key in a database table identifies one record or another. DDS samples of different instances have different values for the key. DDS samples of the same instance of a *Topic* have the same key. Note that not all *Topics* have keys. For *Topics* without keys, there is only a single instance of that *Topic*.

4.3.7 Requesters and Repliers

This section describes the Request-Reply communication pattern, which is only available with the RTI Connext DDS Professional, Evaluation, and Basic package types.

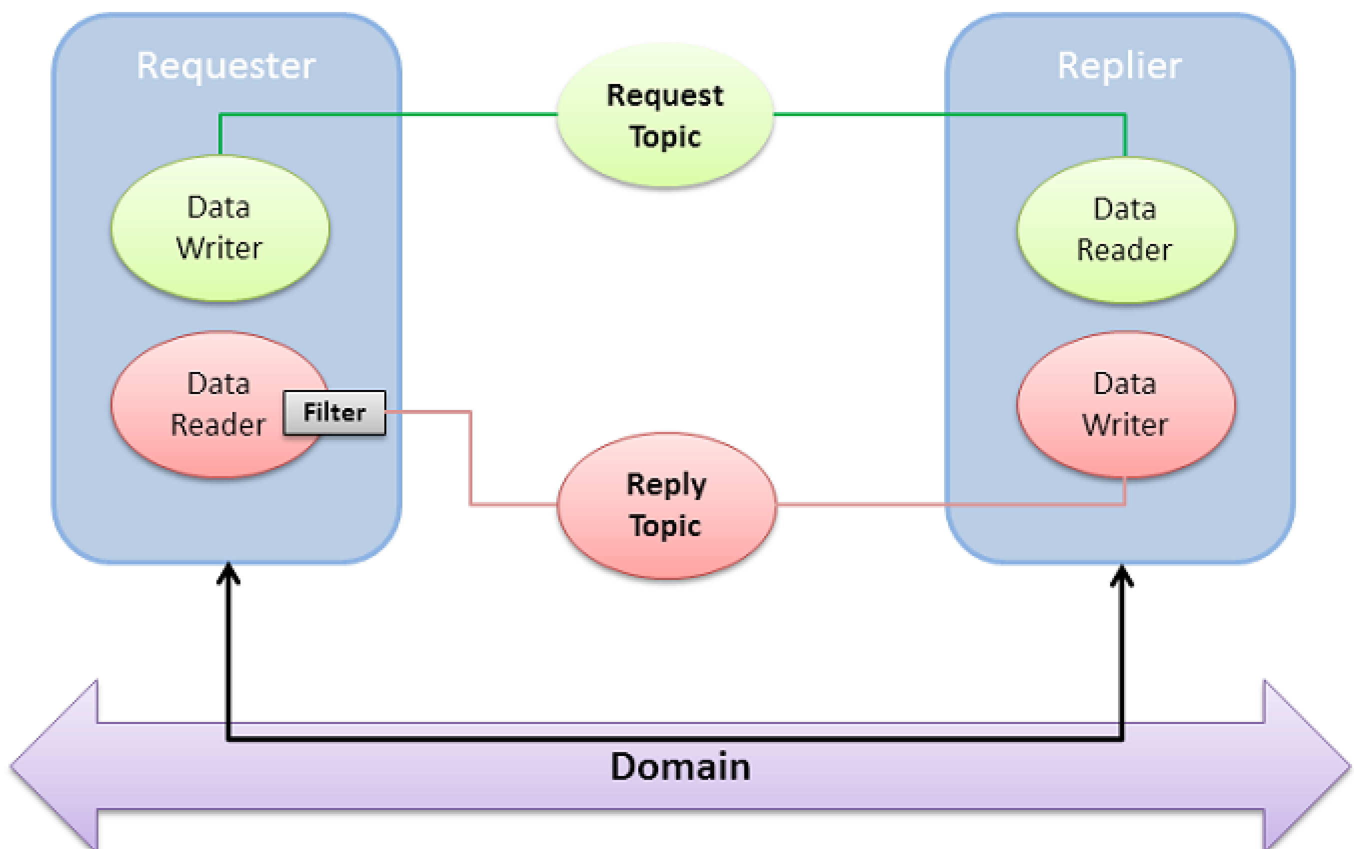
Requesters and *Repliers* provide a way to use the Request-Reply communication pattern on top of the previously described *Connext DDS* entities.

An application uses a *Requester* to send requests to a *Replier*; another application using a *Replier* receives a request and can send one or more replies for that request. The *Requester* that sent the request (and only that one) will receive the reply (or replies).

A *Requester* uses an existing *DomainParticipant* to communicate through a domain. It owns a *DataWriter* for writing requests and a *DataReader* for receiving replies.

Similarly, a *Replier* uses an existing *DomainParticipant* to communicate through a domain and owns a *DataReader* for receiving requests and a *DataWriter* for writing replies.

Figure 4.5: Request-Reply Overview



The Reply Topic filters samples so replies are received by exactly one *Requester*—the one that wrote the related request sample.

You can specify the QoS for the *DataWriters* and *DataReaders* that *Requesters* and *Repliers* create.

The following code shows how to create a *Requester:*

In C++:

```
Requester<Foo, Bar> * requester =
    new Requester<Foo, Bar>(participant, "MyService");
```

In Java:

```
Requester<Foo, Bar> requester = new Requester<Foo, Bar>(
    participant, "MyService",
    FooTypeSupport.get_instance(),
    BarTypeSupport.get_instance());
```

As you can see, we are passing an existing *DomainParticipant* to the constructor.

Foo is the request type and **Bar** is the reply type. In 5.3 Compact, Type-Safe Data Programming with DDS Data Types on page 45, you will learn what types you can use and how to create them.

The constructor also receives a string "MyService." This is the service name, and is used to create the Request Topic and the Reply Topic. In this example, the *Requester* will create a Request Topic called "MyServiceRequest" and a Reply Topic called "MyServiceReply."

Creating a *Replier* is very similar. The following code shows how to create a *Replier*:

In C++:

```
Replier<Foo, Bar> * replier =
    new Replier<Foo, Bar>(participant, "MyService");
```

In Java:

```
Replier<Foo, Bar> replier = new Replier<Foo, Bar>(
    participant, "MyService",
    FooTypeSupport.get_instance(),
    BarTypeSupport.get_instance());
```

This *Replier* will communicate with the *Requester* we created before, because they use the same service name (hence the topics are the same) and they use compatible QoS (the default).

More example code is available for C++, C, Java and C# as part of the API Reference HTML documentation, under **Modules, Programming How-To's, Request-Reply Examples**.

Chapter 5 Capabilities and Performance

In A Quick Overview (Chapter 4 on page 14), you learned the basic concepts in *Connext DDS* and applied them to a simple "Hello, World" application. In this section, you will learn more about some of the powerful and unique benefits of *Connext DDS*:

- A rich set of functionality, implemented for you by the middleware so that you don't have to build it into your application. Most of this functionality—including sophisticated data filtering and expiration, support for durable historical data, and built-in support for periodic data and deadline enforcement—can be defined partially or even completely in declarative quality-of-service (QoS) policies specified in an XML file, allowing you to examine and update your application's configuration without rebuilding or redeploying it. See 5.2 Customizing Behavior: QoS Configuration on page 42 for more information about how to configure QoS policies. Design Patterns for High Performance (Chapter 7 on page 80) describes how to reduce, filter, and cache data as well as other common functional design patterns.
- Compact, type-safe data. The unique and expressive data-typing system built into *Connext DDS* supports not only opaque payloads but also highly structured data. It provides both static and dynamic type safety—without the network overhead of the "self-describing" messages of other networking middleware implementations. See 5.3 Compact, Type-Safe Data Programming with DDS Data Types on page 45 for more information.
- Industry-leading performance. *Connext DDS* provides industry-leading latency, throughput, and jitter performance. Design Patterns for High Performance (Chapter 7 on page 80) provides specific QoS configuration examples to help you get started. You can quickly see the effects of the changes you make using the code examples described in that section. You can benchmark the performance of *Connext DDS* on your own systems with the RTI Example Performance Test. You can download the Example Performance Test from http://www.rti.com/examples/.
- You can also review the data from several performance benchmarks here: http://www.rti.com/products/dds/benchmarks.html. Updated results for new releases are typically published within two months after general availability of that release.

5.1 Automatic Application Discovery

As you've been running the code example described in this guide, you may have noticed that you have not had to start any server processes or configure any network addresses. Its built-in automatic discovery capability is one important way in which *Connext DDS* differs from other networking middleware implementations. It is designed to be low-overhead and require minimal configuration, so in many cases there is nothing you need to do; it will just work. Nevertheless, it's helpful to understand the basics so that you can decide if and when a custom configuration is necessary.

Before applications can communicate, they need to "discover" each other. By default, *Connext DDS* applications discover each other using shared memory or UDP loopback if they are on the same host or using multicast[a]if they are on different hosts. Therefore, to run an application on two or more computers using multicast, or on a single computer with a network connection, no changes are needed. They will discover each other automatically! The section on Discovery in the *RTI Connext DDS Core Libraries User's Manual* describes the process in more detail.

If you want to use computers that do not support multicast (or you need to use unicast for some other reason), or if you want to run on a single computer that does not have a network connection (in which case your operating system may have disabled your network stack), there is a simple way to control the discovery process—you won't even have to recompile. Application discovery can be configured through the NDDS_DISCOVERY_PEERS environment variable or in your QoS configuration file.

5.1.1 When to Set the Discovery Peers

There are only a few situations in which you *must* set the discovery peers:

(In the following, replace *N* with the number of *Connext DDS* applications you want to run.)

1. If you cannot use multicast[b]**:**

   Set your discovery peers to a list of all of the hosts that need to discover each other. The list can contain host names and/or IP addresses; each entry should be of the form ***N*@built-in.udpv4://<hostname|IP>**.

2. If you do not have a network connection:

   Some operating systems—for example, Microsoft Windows—disable some functionality of their network stack when they detect that no network interfaces are available. This can cause problems when applications try to open network connections.

---

[a]With the exception of LynxOS. On LynxOS systems, multicast is not used for discovery by default unless NDDS_DISCOVERY_PEERS is set.

[b]To see if your platform supports multicast, see the *RTI Connext DDS Core Libraries Platform Notes*.

If your system supports shared memory[a], set your discovery peers to ***N*@builtin.shmem://**. This will enable the shared memory transport only.

If your system does *not* support shared memory (or it is disabled), set your discovery peers to the loop-back address, ***N*@builtin.udpv4://127.0.0.1**.

5.1.2 How to Set Your Discovery Peers

As stated above, in most cases you do not need to set your discovery peers explicitly.

If setting them *is* required, there are two easy ways to do so:

- Set the **NDDS_DISCOVERY_PEERS** environment variable to a comma-separated list of the names or addresses of all the hosts that need to discover each other.
  - **Example on Windows systems:**

    ```
    set NDDS_DISCOVERY_PEERS=3@builtin.udpv4://mypeerhost1,
    4@builtin.udpv4://mypeerhost2
    ```

  - **Example on UNIX-based systems when using csh or tcsh:**

    ```
    setenv NDDS_DISCOVERY_PEERS 3@builtin.udpv4://mypeerhost1,
    4@builtin.udpv4://mypeerhost2
    ```

- Set the discovery peer list in your XML QoS configuration file.

[a]To see if your platform supports RTI's shared memory transport, see the *RTI Connext DDS Core Libraries Platform Notes*.

For example, to turn on shared memory only:

```
<participant_qos>
   <discovery>
      <!--
      The initial_peers list are those "addresses" to which the
      middleware will send discovery announcements.
      -->
      <initial_peers>
         <element>4@builtin.shmem://</element>
      </initial_peers>
      <!--
      The multicast_receive_addresses list identifies where the
      DomainParticipant listens for multicast announcements
      from others. Set this list to an empty value to disable
      listening over multicast.
      -->
      <multicast_receive_addresses>
         <!-- empty -->
      </multicast_receive_addresses>
   </discovery>
   <transport_builtin>
      <!--
      The transport_builtin mask identifies which builtin
      transports the DomainParticipant uses. The default value
      is UDPv4 | SHMEM, so set this mask to SHMEM to prevent
      other nodes from initiating communication with this node
      via UDPv4.
      -->
      <mask>SHMEM</mask>
   </transport_builtin>
   ...
</participant_qos>
```

For more information, please see the *Connext DDS Platform Notes*, *User's Manual*, and API Reference HTML documentation (from the main page, select **Modules, Infrastructure Module, QoS Policies, DISCOVERY**).

5.2 Customizing Behavior: QoS Configuration

Almost every object in the *Connext DDS* API is associated with QoS policies that govern its behavior. These policies govern everything from the amount of memory the object may use to store incoming or outgoing data, to the degree of reliability required, to the amount of meta-traffic that *Connext DDS* will send on the network, and many others. The following is a short summary of just a few of the available policies:

- **Reliability and Availability for Consistent Behavior with Critical Data:**
  - **Reliability:** Specifies whether or not the middleware will deliver data reliably. The reliability of a connection between a *DataWriter* and *DataReader* is entirely user configurable. It can be done on a per *DataWriter*-*DataReader* connection.

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 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 Ethernet.

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 the middleware 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.

- **History:** Specifies how much data must be stored by the middleware for the *DataWriter* or *DataReader*. This QoS policy affects the Reliability and Durability QoS policies.

  When a *DataWriter* sends data or a *DataReader* receives data, the data sent or received is stored in a cache whose contents are controlled by the History QosPolicy. The History QosPolicy can tell the middleware to store all of the data that was sent or received, or only store the last *n* values sent or received. If the History QosPolicy is set to keep the last *n* values only, then after *n* values have been sent or received, any new data will overwrite the oldest data in the queue. The queue thus acts like a circular buffer of length *n*.

  This QoS policy interacts with the Reliability QosPolicy by controlling whether or not the middleware guarantees that (a) all of the data sent is received (using the KEEP_ALL setting of the History QosPolicy) or (b) that only the last *n* data values sent are received (a reduced level of reliability, using the KEEP_LAST setting of the History QosPolicy). See the Reliability QosPolicy for more information.

  Also, the amount of data sent to new *DataReaders* whose Durability QosPolicy (see below) is set to receive previously published data is controlled by the History QosPolicy.

- **Lifespan:** Specifies how long the middleware should consider data sent by a user application to be valid.

  The middleware attaches timestamps to all data sent and received. When you specify a finite Lifespan for your data, the middleware will compare the current time with those timestamps and drop data when your specified Lifespan expires. You can 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."

- **Durability:** Specifies whether or not the middleware will store and deliver previously published data to new *DataReaders*. This policy helps ensure that *DataReaders* get all data that was sent by *DataWriters*, even if it was sent while the *DataReader* was disconnected from the network. It can increase a system's tolerance to failure conditions.

- **Fault Tolerance for increased robustness and reduced risk:**
  - **Liveliness:** Specifies and configures the mechanism that allows *DataReaders* to detect when *DataWriters* become disconnected or "dead." It can be used during system integration to ensure that systems meet their intended responsiveness specifications. It can also be used during run time to detect possible losses of connectivity.
  - **Ownership and Ownership Strength:** Along with Ownership Strength, Ownership specifies if a *DataReader* can receive data of a given instance from multiple *DataWriters* at the same time. By default, *DataReaders* for a given topic can receive data of all instances from any *DataWriter* for the same topic. But you can also configure a *DataReader* to receive data of a given instance from only one *DataWriter* at a time. The *DataWriter* with the highest Ownership Strength value will be the owner of the instance and the one whose data is delivered to *DataReaders*. Data of that instance sent by all other *DataWriters* with lower Ownership Strength will be dropped by the middleware.

    When the *DataWriter* with the highest Ownership strength loses its liveliness (as controlled by the Liveliness QosPolicy) or misses a deadline (as controlled by the Deadline QosPolicy) or whose application quits, dies, or otherwise disconnects, the middleware will change ownership of the topic to the *DataWriter* with the highest Ownership Strength from the remaining *DataWriters*. This QoS policy can help you 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.
- **Built-in Support for Periodic Data:**
  - **Deadline:** For a *DataReader*, this QoS specifies the maximum expected elapsed time between arriving DDS data samples. For a *DataWriter*, it specifies a commitment to publish DDS samples with no greater than this elapsed time between them.

    This policy can be used during system integration to ensure that applications have been coded to meet design specifications. It can be used 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.

You can specify an object's QoS two ways: (a) programmatically, in your application's source code or (b) in an XML configuration file. The same parameters are available, regardless of which way you choose. For complete information about all of the policies available, see Chapter 4 in the *RTI Connext DDS Core Libraries User's Manual* or see the API Reference HTML documentation.

The examples covered in this document are intended to be configured with XML files.

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

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

Default path to the examples:

- macOS systems: **/Users/<*your user name*>/rti_workspace/6.0.1/examples**
- Linux systems: **/home/<*your user name*>/rti_workspace/6.0.1/examples**
- Windows systems: **<*your Windows documents folder*>\rti_workspace\6.0.1\examples**

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

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

You can find several example configurations, called *profiles*, in the directory **examples/connext_dds/qos**. The easiest way to use one of these profile files is to either set the environment variable **NDDS_QOS_PROFILES** to the path of the file you want, or copy that file into your current working directory with the file name **USER_QOS_PROFILES.xml** before running your application.

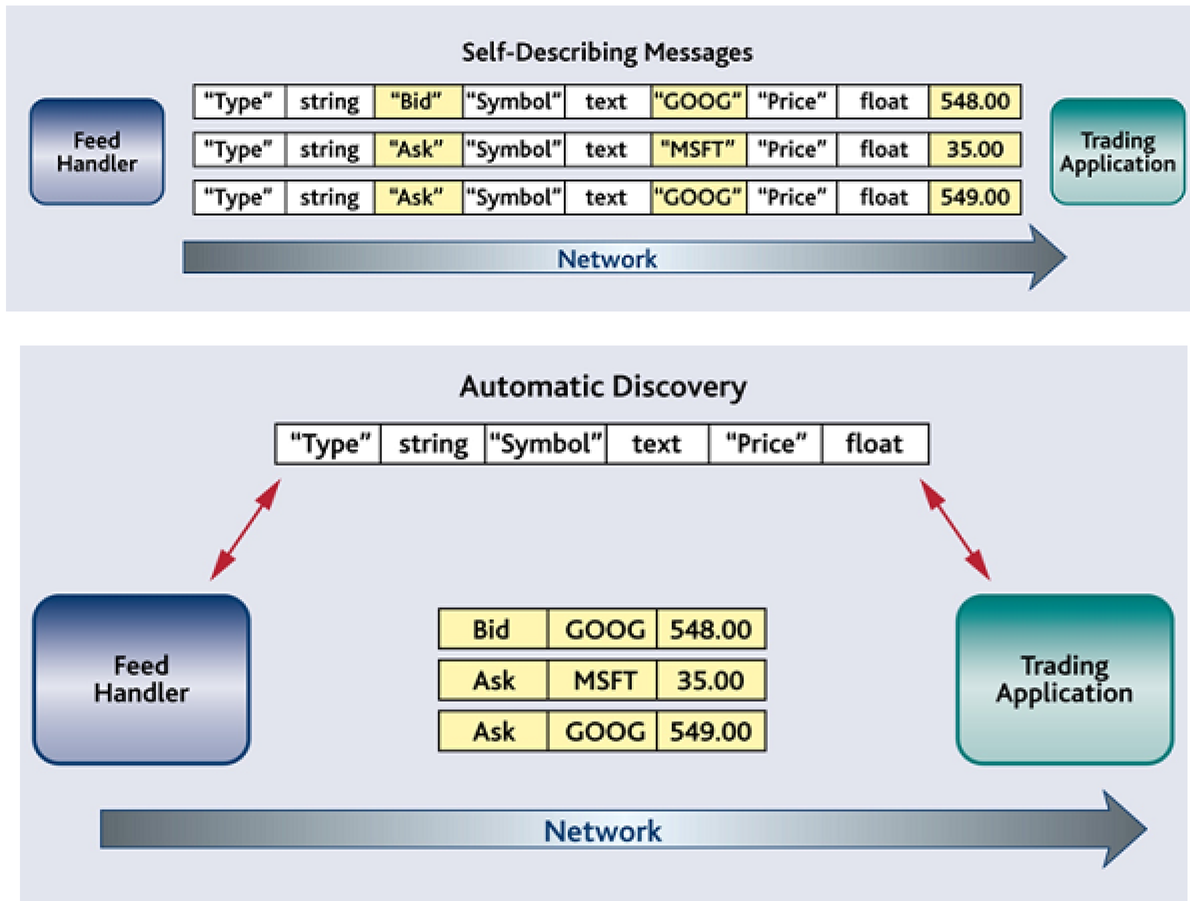
5.3 Compact, Type-Safe Data Programming with DDS Data Types

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/Pentium). This process is commonly called serialization/deserialization or marshalling/demarshalling. Messaging products have typically taken one of two approaches to this problem:

- **Do nothing.** With this approach, the middleware does not provide any help and user code must take into account memory-layout differences when sending messages from one platform to another. The middleware treats messages as an opaque buffer of bytes. The JMS BytesMessage is an example of this approach.
- **Send everything, every time.** Self-describing messages are at the opposite extreme, and embed 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.

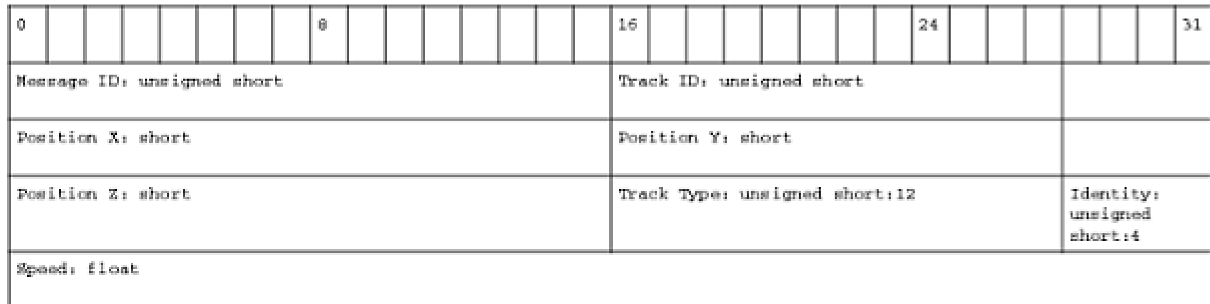
Figure 5.1: Self-Describing Messages vs. Type Definitions



Connext DDS exchanges data type definitions, such as field names and types, once at application start-up time. This increases performance and reduces bandwidth consumption compared to the conventional approach, in which each message is self-describing and thus includes a substantial amount of meta-data along with the actual data.

Connext DDS takes an intermediate approach. Just as objects in your application program belong to some data type, DDS data samples sent on the same *Topic* share a DDS data type. This DDS type defines the fields that exist in the DDS data samples and what their constituent types are; users in the aerospace and defense industries will recognize such type definitions as a form of Interface Definition Document (IDD). *Connext DDS* stores and propagates this meta-information separately from the individual DDS data samples, allowing it to propagate DDS samples efficiently while handling byte ordering and alignment issues for you.

Figure 5.2: Example IDD



This example IDD shows one legacy approach to type definition. RTI supports multiple standard type definition formats that are machine readable as well as human readable.

With RTI, you have a number of choices when it comes to defining and using DDS data types. You can choose one of these options, or you can mix and match them—these approaches interoperate with each other and across programming languages and platforms. So, your options are:

- **Use the built-in DDS types.** If a message is simply a string or a buffer of bytes, you can use RTI's built-in DDS types, described in 5.3.1 Using Built-in DDS Types on the next page.
- **Define a DDS type at compile-time** using a language-independent description language and RTI Code Generator, *rtiddsgen*, as described in 5.3.2 Using DDS Types Defined at Compile Time on the next page. This approach offers the strongest compile-time type safety.

  Whereas in-house middleware implementation teams often define data formats in word processing or spreadsheet documents and translate those formats into application code by hand, RTI relies on standard type definition formats that are both human- and machine-readable and generates code in compliance with open international standards. The code generator accepts data-type definitions in a number of formats to make it easy to integrate *Connext DDS* 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 the *RTI Connext DDS Core Libraries User's Manual*.
  - *XML schema* (XSD), whether independent or embedded in a WSDL file. XSD may be the format of choice for those using *Connext DDS* alongside or connected to a web services infrastructure. This format is described in *RTI Connext DDS Core Libraries User's Manual*.
  - *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 the *RTI Connext DDS Core Libraries User's Manual*.

- **Define a dynamic type programmatically at run time.**[a] 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 allows you to use an API similar to those of Tibco Rendezvous or JMS MapMessage to manipulate messages without sacrificing efficiency. It is described in 5.3.4 Running with Dynamic DDS Types on page 56.

The following sections of this document describe each of these models.

5.3.1 Using Built-in DDS Types

Connext DDS provides a set of standard types that are built into the middleware. These DDS types can be used immediately. The supported built-in DDS types are **String**, **KeyedString**, **Octets**, and **KeyedOctets**. (On Java and .NET platforms, the latter two types are called **Bytes** and **KeyedBytes**, respectively; the names are different but the data is compatible across languages.) **String** and **KeyedStrings** can be used to send variable-length strings of single-byte characters. **Octets** and **KeyedOctets** can be used to send variable-length arrays of bytes.

These built-in DDS types may be sufficient if your data-type needs are simple. If your data is more complex and highly structured, or you want *Connext DDS* to examine fields within the data for filtering or other purposes, this option may not be appropriate, and you will need to take additional steps to use compile-time types (see 5.3.2 Using DDS Types Defined at Compile Time below) or dynamic types (see 5.3.4 Running with Dynamic DDS Types on page 56).

5.3.2 Using DDS Types Defined at Compile Time

In this section, we define a DDS type at compile time using a language-independent description and RTI Code Generator (*rtiddsgen*).

RTI Code Generator accepts data-type definitions in a number of formats, such as OMG IDL, XML Schema (XSD), and a DDS-specific format of XML. This makes it easy to integrate *Connext DDS* with your development processes and IT infrastructure. In this section, we will define DDS types using IDL. (In case you would like to experiment with a different format, *rtiddsgen* can convert from any supported format to any other: simply pass the arguments **-convertToIdl**, **-convertToXml**, **-convertToXsd**, or **-convertToWsdl**.)

As described in the *Release Notes*, some platforms are supported as both a host and a target, while others are only supported as a target. The *rtiddsgen* tool must be run on a computer that is supported as a host. For target-only platforms, you will need to run *rtiddsgen* and build the application on a separate host computer.

The following sections will take your through the process of generating example code from your own data type.

---

[a] Dynamic types are not supported when using the separate add-on product, RTI Ada Language Support.

5.3.3 Generating Code with RTI Code Generator

Don't worry about how DDS types are defined for now. For this example, just copy and paste the following into a new file, **HelloWorld.idl.**

```
const long HELLO_MAX_STRING_SIZE = 256;
struct HelloWorld {
       string<HELLO_MAX_STRING_SIZE> message;
};
```

Next, we will invoke *RTI Code Generator* (*rtiddsgen*), which can be found in the **$NDDSHOME/bin** directory that should already be on your path, to create definitions of your data type in a target programming language, including logic to serialize and deserialize instances of that type for transmission over the network. Then we will build and run the generated code.

For a complete list of the arguments *rtiddsgen* understands, and a brief description of each of them, run it with the **-help** argument. More information about *rtiddsgen*, including its command-line parameters and the list of files it creates, can be found in the *RTI Connext DDS Core Libraries User's Manual*.

Note: Running *rtiddsgen* on a Red Hat Enterprise Linux 4.0 target platform is not supported because it uses an older JRE. You can, however, run *rtiddsgen* on a newer Linux platform to generate code that can be used on the Red Hat Enterprise Linux 4.0 target.

Instructions for Traditional C++

Generate C++ code from your IDL file with the following command (replace the architecture name**i86Linux3gcc4.8.2** with the name of your own architecture):

```
> rtiddsgen -ppDisable              \
       -language C++                \
       -example i86Linux3gcc4.8.2   \
       -replace                     \
       HelloWorld.idl
```

The generated code publishes identical DDS data samples and subscribes to them, printing the received data to the terminal. Edit the code to modify each DDS data sample before it's published: Open **HelloWorld_publisher.cxx**. In the code for **publisher_main()**, locate the "for" loop and add the bold line seen below, which puts "Hello World!" and a consecutive number in each DDS sample that is sent.[a]

[a] If you are using Visual Studio, consider using sprintf_s instead of sprintf: sprintf_s(instance->msg, 128, "Hello World! (%d)", count);

```
for (count=0; (sample_count == 0) || (count < sample_count); ++count) {
       printf("Writing HelloWorld, count %d\n", count);
       /* Modify the data to be sent here */
               sprintf(instance->message, "Hello World! (%d)", count);

       retcode = HelloWorld_writer->write(*instance, instance_handle);
       if (retcode != DDS_RETCODE_OK) {
               printf("write error %d\n", retcode);
       }
       NDDSUtility::sleep(send_period);
}
```

Instructions for Modern C++

Generate C++03 or C++11 code from your IDL file with the following command (replace the architecture name **i86Linux3gcc4.8.2** with the name of your own architecture):

```
> rtiddsgen -ppDisable              \
       -language C++03             \
       -example i86Linux3gcc4.8.2   \
       -replace                    \
       HelloWorld.idl
```

Or:

```
> rtiddsgen -ppDisable             \
       -language C++11             \
       -example i86Linux3gcc4.8.2   \
       -replace                    \
       HelloWorld.idl
```

The only differences between using C++03 or C++11 is that the latter creates a different subscriber example and adds a flag to enable C++11 in supported compilers that require explicit activation. This also activates C++11 features in the DDS API headers. See the API Reference HTML documentation (Modules > Conventions) for more information.

The generated code publishes identical DDS data samples and subscribes to them, printing the received data to the terminal. Edit the code to modify each DDS data sample before it's published: Open **HelloWorld_publisher.cxx**. In the code for **publisher_main()**, locate the "for" loop and add the bold line seen below, which puts "Hello World!" and a consecutive number in each DDS sample that is sent.[a]

[a] If you are using Visual Studio, consider using sprintf_s instead of sprintf: sprintf_s(instance->msg, 128, "Hello World! (%d)", count);

```
for (int count = 0; count < sample_count || sample_count == 0; count++) {
    // Modify the data to be written here
    sample.message("Hello World! (" + std::to_string(count) + ")");
    std::cout << "Writing MyOtherType, count " << count << "\n";
    writer.write(sample);
    rti::util::sleep(dds::core::Duration(4));
}
```

Instructions for Java

Generate Java code from your IDL file with the following command[a] (replace the architecture name **i86Linux3gcc4.8.2** with the name of your own architecture):

```
> rtiddsgen -ppDisable              \
       -language Java               \
       -example  i86Linux3gcc4.8.2 \
       -replace                     \
       HelloWorld.idl
```

The generated code publishes identical DDS data samples and subscribes to them, printing the received data to the terminal. Edit the code to modify each DDS data sample before it's published: Open **HelloWorldPublisher.java**. In the code for **publisherMain()**, locate the "for" loop and add the bold line seen below, which puts "Hello World!" and a consecutive number in each DDS sample that is sent.

```
for
(int count = 0; sampleCount == 0) || (count < sampleCount); ++count) {
       System.out.println("Writing HelloWorld, count " + count);
       /* Modify the instance to be written here */
       instance.msg = "Hello World! (" + count + ")";
       /* Write data */
       writer.write(instance, InstanceHandle_t.HANDLE_NIL);
       try {
               Thread.sleep(sendPeriodMillis);
       } catch (InterruptedException ix) {
               System.err.println("INTERRUPTED");
               break;
       }
}
```

Instructions for Ada

Generate Ada code from your IDL file with the following command (replace the architecture name x64Linux2.6gcc4.4.5 with the name of your own architecture):

---

[a]The argument -ppDisable tells the code generator not to attempt to invoke the C preprocessor (usually cpp on UNIX systems and cl on Windows systems) on the IDL file prior to generating code. In this case, the IDL file contains no preprocessor directives, so no preprocessing is necessary. However, if the preprocessor executable is already on your system path (on Windows systems, running the Visual Studio script vcvars32.bat will do this for you) you can omit this argument.

```
rtiddsgen -ppDisable -language Ada \
   -example x64Linux2.6gcc4.4.5 -replace HelloWorld.idl
```

Notes:

- Ada support requires a separate add-on product, *Ada Language Support*.
- The generated publisher and subscriber source files are under the **samples** directory. There are two generated project files: one at the top level and one in the **samples** directory. The project file in the samples directory, **samples/helloworld-samples.gp**r, should be the one that you will use to compile the example.
- The generated Ada project files need two directories to compile a project: **.obj** and **bin**. If your Ada IDE does not automatically create these directories, you will need to create them outside the Ada IDE, in both the top-level directory and in **samples** directory.

```
for the "Count in 0 .. Sample_Count "loop
   Put_Line ("Writing HelloWorld, count " & Count'Img);
       declare
           Msg : DDS.String := DDS.To_DDS_String
                ("Hello World! (" &    Count'Img & ")");
        begin
            if Instance.message /= DDS.NULL_STRING then
                Finalize (Instance.message);
            end if;
            Standard.DDS.Copy(Instance.message, Msg);
        end;
        HelloWorld_Writer.Write (
            Instance_Data => Instance,
            Handle => Instance_Handle'Unchecked_Access);
        delay Send_Period;
end loop;
```

5.3.3.1 Building the Generated Code

You have now defined your data type, generated code for it, and customized that code. It's time to compile the example applications.

Building a Generated C, C++, or .NET Example on Windows Systems

With the NDDSHOME environment variable set, start Visual Studio and open the rtiddsgen-generated solution file (**.sln**). Select the Release configuration in the Build toolbar in Visual Studio[a]. From the Build menu, select Build Solution. This will build two projects: **<IDL name>_publisher** and **<IDL name>_ subscriber**.

Building a Generated Ada Example on a Linux System

Ada support requires a separate add-on product, *Ada Language Support*.

[a]The *Connext DDS* .NET language binding is currently supported for C# and C++/CLI.

Use the generated Ada project file to compile an example on any system.

Note: The generated project file assumes the correct version of the compiler is already on your path, NDDSHOME is set, and **$NDDSHOME/lib/gnat** is in your ADA_PROJECT_PATH.

```
gmake -f makefile_HelloWorld_<architecture>
```

After compiling the Ada example, you will find the application executables in the directory, **samples/bin**. The build command in the generated makefile uses the static release versions of the Connext DDS. To select dynamic or debug versions of the libraries, change the Ada compiler variables RTIDDS_ LIBRARY_TYPE and RTIDDS_BUILD in the build command in the makefile to build with the desired version of the libraries. For example, if the application must be compiled with the relocatable debug version of the libraries, compile with the following command:

```
gprbuild -p -P samples/helloworld-samples.gpr -XOS=Linux \
  -XRTIDDS_LIBRARY_TYPE=relocatable -XRTIDDS_BUILD=debug -XARCH=${ARCH}
```

Building a Generated Example on Other Platforms

Use the generated makefile to compile a C or C++ example on a UNIX-based system or a Java example on any system. Note: The generated makefile assumes the correct version of the compiler is already on your path and that NDDSHOME is set. If you do not have **gmake** on your system, you can copy the lines from the generated makefile to compile your example.

```
gmake -f makefile_HelloWorld_<architecture>
```

After compiling the C or C++ example, you will find the application executables in a directory **objs/<architecture>**.

The generated makefile includes the static release versions of the *Connext DDS*. To select dynamic or debug versions of the libraries, edit the makefile to change the library suffixes. Generally, *Connext DDS* uses the following convention for library suffixes: "z" for static release, "zd" for static debug, none for dynamic release, and "d" for dynamic debug. For a complete list of the required libraries for each configuration, see the *Connext DDS Platform Notes.*

For example, to change a C++ makefile from using static release to dynamic release libraries, change this directive:

```
LIBS = -L$(NDDSHOME)/lib/<architecture> \
       -lnddscppz -lnddscz -lnddscorez $(syslibs_<architecture>)
```

to:

```
LIBS = -L$(NDDSHOME)/lib/<architecture> \
       -lnddscpp -lnddsc -lnddscore $(syslibs_<architecture>)
```

5.3.3.2 Running the Example Applications

Run the example publishing and subscribing applications and see them communicate:

Running the Generated C++ Example

First, start the subscriber application, **HelloWorld_subscriber**:

```
./objs/<architecture>/HelloWorld_subscriber
```

In this command window, you should see that the subscriber wakes up every four seconds to print a message:

```
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
```

Next, open another command prompt window and start the publisher application, **HelloWorld_publisher**. For example:

```
./objs/<architecture>/HelloWorld_publisher
```

In this second (publishing) command window, you should see:
Writing HelloWorld, count 0 Writing HelloWorld, count 1 Writing HelloWorld, count 2

Look back in the first (subscribing) command window. You should see that the subscriber is now receiving messages from the publisher:

```
HelloWorld subscriber sleeping for 4 sec...
       msg: "Hello World! {0}"
HelloWorld subscriber sleeping for 4 sec...
       msg: "Hello World! {1}"
HelloWorld subscriber sleeping for 4 sec...
       msg: "Hello World! {2}"
```

Running the Generated Java Example

You can run the generated applications using the generated makefile. On most platforms, the generated makefile assumes the correct version of **java** is already on your path and that the NDDSHOME environment variable is set[a].

First, run the subscriber:

[a]One exception on LynxOS systems; see *Getting Started on Embedded UNIX-like Systems* in the *Getting Started Guide, Addendum for Embedded Platforms.*

```
gmake -f makefile_HelloWorld_<architecture> HelloWorldSubscriber
```

In this command window, you should see that the subscriber wakes up every four seconds to print a message:

```
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
HelloWorld subscriber sleeping for 4 sec...
```

Next, run the publisher:

```
gmake -f makefile_HelloWorld_<architecture> HelloWorldPublisher
```

In this second (publishing) command window, you should see:

```
Writing HelloWorld, count 0
Writing HelloWorld, count 1
Writing HelloWorld, count 2
```

Look back in the first (subscribing) command window. You should see that the subscriber is now receiving messages from the publisher:

```
HelloWorld subscriber sleeping for 4 sec...
   msg: "Hello World! {0}"
HelloWorld subscriber sleeping for 4 sec...
   msg: "Hello World! {1}"
HelloWorld subscriber sleeping for 4 sec...
   msg: "Hello World! {2}"
```

Running the Generated Ada Example

Ada support requires a separate add-on product, *Ada Language Support*.

First, start the subscriber application, **helloworld_idl_file-helloworld_subscriber**:

```
./samples/bin/helloworld_idl_file-helloworld_subscriber
```

In this command window, you should see that the subscriber wakes up every four seconds to print a message:

```
HelloWorld subscriber sleeping for 4.000000000 sec.
HelloWorld subscriber sleeping for 4.000000000 sec.
HelloWorld subscriber sleeping for 4.000000000 sec.
```

Next, open another command prompt window and start the publisher application, helloworld_idl_file-helloworld_publisher. For example:

```
./samples/bin/helloworld_idl_file-helloworld_publisher
```

In this second (publishing) command window, you should see:

```
Writing HelloWorld, count 0
Writing HelloWorld, count 1
Writing HelloWorld, count 2
```

Look back in the first (subscribing) command window. You should see that the subscriber is now receiving messages from the publisher:

```
HelloWorld subscriber sleeping for 4 sec...
       msg: "Hello World! {0}"
HelloWorld subscriber sleeping for 4 sec...
       msg: "Hello World! {1}"
HelloWorld subscriber sleeping for 4 sec...
       msg: "Hello World! {2}"
```

5.3.4 Running with Dynamic DDS Types

Dynamic DDS types are not supported when using Ada Language Support

This method may be appropriate for applications for which the structure (type) of messages changes frequently or for deployed systems in which newer versions of applications need to interoperate with existing applications that cannot be recompiled to incorporate message-type changes.

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.

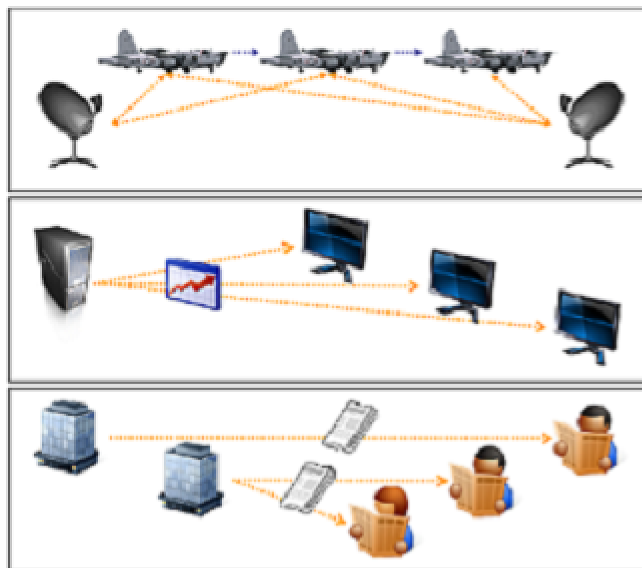
While covering dynamic DDS types is outside the scope of this section, you can learn more about the subject in Chapter 3 of the *RTI Connext DDS Core Libraries User's Manual*. You can also view and run the Hello World example code located in **examples/connext_dds/<language>/Hello_dynamic/src**.

For the examples in Modern C++, see the Modern C++ API reference, Modules > Programming How-To's > DynamicType and DynamicData Use Cases.

Chapter 6 Design Patterns for Rapid Development

In this section, you will learn how to implement some common functional design patterns. As you have learned, one of the advantages to using *Connext DDS* is that you can achieve significantly different functionality without changing your application code simply by updating the XML-based Quality of Service (QoS) parameters.

In this section, we will look at a simple newspaper example to illustrate these design patterns. Newspaper distribution has long been a canonical example of publish-subscribe communication, because it provides a simple metaphor for real-world problems in a variety of industries.



A radar tracking system and a market data distribution system share many features with the news subscriptions we are all familiar with: many-to-many publish-subscribe communication with certain quality-of-service requirements.

In a newspaper scenario (provided in an example called **news** example for all languages, a news publishing application distributes articles from a variety of news outlets—CNN, Bloomberg, etc.—

on a periodic basis. However, the period differs from outlet to outlet. One or more news subscribers poll for available articles, also on a periodic basis, and print out their contents. Once published, articles remain available for a period of time, during which subscribing applications can repeatedly access them if they wish. After that time has elapsed, the middleware will automatically expire them from its internal cache.

This section describes 6.1 Building and Running the News Examples below and includes the following design patterns:

- 6.2 Subscribing Only to Relevant Data on page 60
- 6.3 Accessing Historical Data when Joining the Network on page 69
- 6.4 Caching Data within the Middleware on page 71
- 6.5 Receiving Notifications When Data Delivery Is Late on page 75

You can find more examples at http://www.rti.com/examples. This page contains example code snippets on how to use individual features, examples illustrating specific use cases, as well as performance test examples.

6.1 Building and Running the News Examples

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

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

Default path to the examples:

- macOS systems: **/Users/<*your user name*>/rti_workspace/6.0.1/examples**
- Linux systems: **/home/<*your user name*>/rti_workspace/6.0.1/examples**
- Windows systems: **<*your Windows documents folder*>\rti_workspace\6.0.1\examples**

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

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

Source code for the **news** example is in **<path to examples>/connext_dds/<language>/news**.

The example performs these steps:

1. Parses the command-line arguments.

2. Loads the quality-of-service (QoS) file, **USER_QOS_PROFILES.xml**, from the current working directory. The middleware does this automatically; you will not see code in the example to do this. For more information on how to use QoS profiles, see the section on "Configuring QoS with XML" in the *RTI Connext DDS Core Libraries User's Manual*.

3. On the publishing side, sends news articles periodically.

4. On the subscribing side, receives these articles and print them out periodically.

The steps for compiling and running the program are similar to those described in 4.1 Building and Running "Hello, World" on page 14. As with the Hello World example, Java users should use the build and run command scripts in **<path to examples>/connext_dds/java/news**. Non-Java Windows developers will find the necessary Microsoft Visual Studio solution files in **<path to examples>/connext_dds/<language>/news/win32**. Appropriate makefiles for building the C, C++, or Ada[a] examples on UNIX platforms are in **<path to examples>/connext_dds/<language>/news/make**.

The **news** example combines the publisher and subscriber in a single program, which is run from the **<path to examples>/connext_dds/<language>/news** directory with the argument **pub** or **sub** to select the desired behavior. When running with default arguments, you should see output similar to that shown in 4.1 Building and Running "Hello, World" on page 14. To see additional command-line options, run the program without any arguments.

Both the publishing application and the subscribing application operate in a periodic fashion:

The publishing application writes articles from different news outlets at different rates. Each of these articles is numbered consecutively with respect to its news outlet. Every two seconds, the publisher prints a summary of what it wrote in the previous period.

The subscribing application polls the cache of its **DataReader** every two seconds and prints the data it finds there. (Many real-world applications will choose to process data as soon as it arrives rather than polling for it. For more information about the different ways to read data, select **Modules, Programming How-To's, DataReader Use Cases** in the API Reference HTML documentation. In this case, periodic polling makes the behavior easy to illustrate.) Along with each article it prints, it includes the time at which that article was published and whether that article has been read before or whether it was cached from a previous read.

By default, both the publishing and subscribing applications run for 20 seconds and then quit. (To run them for a different number of seconds, use the **-r** command-line argument.) Figure 6.1: Example Output for Both Applications on the next page shows example output using the default run time and a domain ID of 13 (set with the command-line option, **-d 13**).

---

[a]Ada support requires a separate add-on product, *Ada Language Support*.

Figure 6.1: Example Output for Both Applications

```
Terminal — ssh — 60x45
> ./run.sh pub -d 13
News Example Application
Copyright 2009 Real-Time Innovations, Inc.

Articles published in last period as of 9:50:15 AM:
    From Reuters   : "lorem ipsum 1"
    From AP        : "lorem ipsum 1"
    From CNN       : "lorem ipsum 1"
    From Bloomberg : "lorem ipsum 1"
    From NY Times  : "lorem ipsum 1"
    From Economist : "lorem ipsum 1"
    From Reuters   : "lorem ipsum 2"
    From Reuters   : "lorem ipsum 3"
    From AP        : "lorem ipsum 2"
    From Reuters   : "lorem ipsum 4"
    From CNN       : "lorem ipsum 2"
    From Reuters   : "lorem ipsum 5"
    From AP        : "lorem ipsum 3"
    From Bloomberg : "lorem ipsum 2"
    From Reuters   : "lorem ipsum 6"
    From NY Times  : "lorem ipsum 2"
    From Reuters   : "lorem ipsum 7"
    From AP        : "lorem ipsum 4"
    From CNN       : "lorem ipsum 3"
    From Economist : "lorem ipsum 2"
    From Reuters   : "lorem ipsum 8"
    From Reuters   : "lorem ipsum 9"
    From AP        : "lorem ipsum 5"
    From Bloomberg : "lorem ipsum 3"
    From Reuters   : "lorem ipsum 10"
    From CNN       : "lorem ipsum 4"
    From Reuters   : "lorem ipsum 11"
    From AP        : "lorem ipsum 6"
    From NY Times  : "lorem ipsum 3"
Articles published in last period as of 9:50:17 AM:
    From Reuters   : "lorem ipsum 12"
    From Reuters   : "lorem ipsum 13"
    From AP        : "lorem ipsum 7"
    From CNN       : "lorem ipsum 5"
    From Bloomberg : "lorem ipsum 4"
    From Economist : "lorem ipsum 3"
    From Reuters   : "lorem ipsum 14"
    From Reuters   : "lorem ipsum 15"
    From AP        : "lorem ipsum 8"
    From Reuters   : "lorem ipsum 16"
```

```
Terminal — ssh — 60x45
> ./run.sh sub -d 13
News Example Application
Copyright 2009 Real-Time Innovations, Inc.

Available articles as of 9:50:14 AM:
    [9:50:13 AM] From Reuters   : "lorem ipsum 1"
    [9:50:13 AM] From AP        : "lorem ipsum 1"
    [9:50:13 AM] From CNN       : "lorem ipsum 1"
    [9:50:13 AM] From Bloomberg : "lorem ipsum 1"
    [9:50:13 AM] From NY Times  : "lorem ipsum 1"
    [9:50:13 AM] From Economist : "lorem ipsum 1"
    [9:50:13 AM] From Reuters   : "lorem ipsum 2"
    [9:50:14 AM] From Reuters   : "lorem ipsum 3"
    [9:50:14 AM] From AP        : "lorem ipsum 2"
    [9:50:14 AM] From Reuters   : "lorem ipsum 4"
    [9:50:14 AM] From CNN       : "lorem ipsum 2"
Available articles as of 9:50:16 AM:
    [9:50:13 AM] From AP        : "lorem ipsum 1" (cached)
    [9:50:13 AM] From CNN       : "lorem ipsum 1" (cached)
    [9:50:13 AM] From Bloomberg : "lorem ipsum 1" (cached)
    [9:50:13 AM] From NY Times  : "lorem ipsum 1" (cached)
    [9:50:13 AM] From Economist : "lorem ipsum 1" (cached)
    [9:50:14 AM] From AP        : "lorem ipsum 2" (cached)
    [9:50:14 AM] From CNN       : "lorem ipsum 2" (cached)
    [9:50:14 AM] From Reuters   : "lorem ipsum 5"
    [9:50:14 AM] From AP        : "lorem ipsum 3"
    [9:50:14 AM] From Bloomberg : "lorem ipsum 2"
    [9:50:14 AM] From Reuters   : "lorem ipsum 6"
    [9:50:14 AM] From NY Times  : "lorem ipsum 2"
    [9:50:14 AM] From Reuters   : "lorem ipsum 7"
    [9:50:14 AM] From AP        : "lorem ipsum 4"
    [9:50:14 AM] From CNN       : "lorem ipsum 3"
    [9:50:14 AM] From Economist : "lorem ipsum 2"
    [9:50:15 AM] From Reuters   : "lorem ipsum 8"
    [9:50:15 AM] From Reuters   : "lorem ipsum 9"
    [9:50:15 AM] From AP        : "lorem ipsum 5"
    [9:50:15 AM] From Bloomberg : "lorem ipsum 3"
    [9:50:15 AM] From Reuters   : "lorem ipsum 10"
    [9:50:15 AM] From CNN       : "lorem ipsum 4"
    [9:50:15 AM] From Reuters   : "lorem ipsum 11"
    [9:50:15 AM] From AP        : "lorem ipsum 6"
    [9:50:15 AM] From NY Times  : "lorem ipsum 3"
    [9:50:15 AM] From Reuters   : "lorem ipsum 12"
    [9:50:16 AM] From Reuters   : "lorem ipsum 13"
    [9:50:16 AM] From AP        : "lorem ipsum 7"
```

6.2 Subscribing Only to Relevant Data

From reading A Quick Overview (Chapter 4 on page 14), you already understand how to subscribe only to the topics in which you're interested. However, depending on your application, you may be interested in only a fraction of the data available on those topics. Extraneous, uninteresting, or obsolete data puts a drain on your network and CPU resources and complicates your application logic.

Fortunately, *Connext DDS* can perform much of your filtering and data reduction for you. *Data reduction* is a general term for discarding unnecessary or irrelevant data so that you can spend your time processing the data you care about. You can define the set of data that is relevant to you based on:

- **Its content**. Content-based filters can examine any field in your data based on a variety of criteria, such as whether numeric values meet various equality and inequality relationship or whether string values match certain regular expression patterns. For example, you may choose to distribute a stream of stock prices using a single topic, but indicate that you're interested only in the price of IBM, and only when that price goes above $20.
- **How old it is**. You can indicate that data is relevant for only a certain period of time (its lifespan) and/or that you wish to retain only a certain number of DDS data samples (a history depth). For

example, if you are interested in the last ten articles from each news outlet, you can set the history depth to 10.

- **How fast it's updated**. Sometimes data streams represent the state of a moving or changing object—for example, an application may publish the position of a moving vehicle or the changing price of a certain financial instrument. Subscribing applications may only be able to—or interested in—processing this data at a certain maximum rate. For example, if the changing state is to be plotted in a user interface, a human viewer is unlikely to be able to process more than a couple of changes per second. If the application attempts to process updates any faster, it will only confuse the viewer and consume resources unnecessarily.

6.2.1 Content-Based Filtering

A *DataReader* can filter incoming data by subscribing not to a given Topic itself, but to a *ContentFilteredTopic* that associates the *Topic* with an SQL-like expression that indicates which DDS data samples are of interest to the subscribing application. Each subscribing application can specify its own content-based filter, as it desires; publishing applications' code does not need to change to allow subscribers to filter on data contents.

6.2.1.1 Implementation

In Traditional C++:

```
DDSContentFilteredTopic cft = participant->create_contentfilteredtopic(
cftName.c_str(),
       topic,
       contentFilterExpression.c_str(),
       noFilterParams);
if (cft == NULL) {
       throw std::runtime_error("Unable to create ContentFilteredTopic");
}
```

The Topic and ContentFilteredTopic classes share a base class: TopicDescription.

The variable **contentFilterExpression** in the above example code is a SQL expression; see below.

In Modern C++:

```
using namespace dds::topic;
Topic<Stock> topic(participant, "My Stock");=
ContentFilteredTopic<Stock> cft(
    topic, "My Stock (filtered)", Filter("key='CNN' OR key='Reuters'");
```

The Topic and ContentFilteredTopic classes share a base class: TopicDescription.

In Java:

```
ContentFilteredTopic cft = participant.create_contentfilteredtopic(
       topic.get_name() + " (filtered)", topic,
       contentFilterExpression, null);
if (cft == null) {
     throw new IllegalStateException(
    "Unable to create ContentFilteredTopic");
}
```

In Ada:

```
declare
cft     : DDS.ContentFilteredTopic.Ref_Access;
cftName : DDS.String := DDS.To_DDS_String (DDS.To_Standard_String(
       topic.Get_Name) & " (filtered)");
begin
       cft := participant.Create_Contentfilteredtopic(
               cftName, topic, contentFilterExpression, null);
       DDS.Finalize (cftName);
       if cft = null then
               Put_Line (Standard_Error,
                       "Unable to create ContentFilteredTopic");
               return;
       end if;
end;
```

In Ada, ContentFilteredTopic class inherits from TopicDescription while Topic has a function 'As_TopicDescription' that is useful to create a *DataReader* where a TopicDescription is needed.

6.2.1.2 Running & Verifying

The following shows a simple filter with which the subscribing application indicates it is interested only in news articles from CNN or Reuters; you can specify such a filter with the **–f** or **--filterExpression** command-line argument, like this in the Java example:

```
> ./run.sh sub -f "key='CNN' OR key='Reuters'"
```

This example uses the built-in **KeyedString** data type. This type has two fields: **key**, a string field that is the type's only key field, and **value**, a second string. This example uses the **key** field to store the news outlet name and the **value** field to store the article text. The word "key" in the content filter expression "key='CNN'" refers to the field's name, not the fact that it is a key field; you can also filter on the value field if you like. In your own data types, you will use the name(s) of the fields you define. For example output, see Figure 6.2: Using a Content-Based Filter on the next page.

You can find more detailed information in the API Reference HTML documentation under **Modules, Connext DDS API Reference, Queries and Filters Syntax**. For Ada, open **<NDDSHOME>/doc/api/connext_dds/api_ada/index.html** and look under **Connext DDS API Reference, DDSQueryAndFilterSyntaxModule**.

Figure 6.2: Using a Content-Based Filter

6.2.2 Lifespan and History Depth

One of the most common ways to reduce data is to indicate for how long a DDS data sample remains valid once it has been sent. You can indicate such a contract in one (or both) of two ways: in terms of a number of DDS samples to retain (the history depth) and the elapsed time period during which a DDS sample should be retained (the lifespan). For example, you may be interested in only the most recent data value (the so-called "last values") or in all data that has been sent during the last second.

The history depth and lifespan, which can be specified declaratively with QoS policies (see below), apply both to durable historical data sent to late-joining subscribing applications as well as to current subscribing applications. For example, suppose a subscribing application has set history depth = 2, indicating that it is only interested in the last two DDS samples. A publishing application sends four DDS data samples in close succession, and the middleware on the subscribing side receives them before the application chooses to read them from the middleware. When the subscribing application does eventually call **DataReader::read()**, it will see only DDS samples 3 and 4; DDS samples 1 and 2 will already have been overwritten. If an application specifies both a finite history depth and a finite lifespan, whichever limit is reached first will take effect.

6.2.2.1 Implementation

History depth is part of the History QoS policy. The lifespan is specified with the Lifespan QoS policy. The History QoS policy applies *independently* to both the *DataReader* and the *DataWriter*; the values specified on both sides of the communication need not agree. The Lifespan QoS policy is specified only on the *DataWriter*, but it is enforced on both sides: the *DataReader* will enforce the lifespan indicated by each *DataWriter* it discovers for each DDS sample it receives from that *DataWriter*.

For more information about these QoS policies, consult the API Reference HTML documentation. Open ReadMe.html and select a programming language, then select **Modules, RTI Connext DDS API Reference, Infrastructure, QoS Policies**. For Ada: open **<NDDSHOME>/doc/api/connext_dds/api_ada/index.html** and select **Infrastructure Module, DDSQosTypesModule**.

You can specify these QoS policies either in your application code or in one or more XML files. Both mechanisms are functionally equivalent; the **News** example provided for C, C++, Java, and Ada uses XML files. For more information about this mechanism, see the section on "Configuring QoS with XML" in the *RTI Connext DDS Core Libraries User's Manual*.

History Depth:

The DDS specification, which *Connext DDS* implements, recognizes two "kinds" of history: **KEEP_ALL** and **KEEP_LAST**. **KEEP_ALL** history indicates that the application wants to see every DDS data sample, regardless of how old it is (subject, of course, to Lifespan and other QoS policies). **KEEP_LAST** history indicates that only a certain number of back DDS samples are relevant; this number is indicated by a second parameter, the history **depth**. (The **depth** value, if any, is ignored if the **kind** is set to KEEP_ALL.)

To specify a default History **kind** of **KEEP_LAST** and a **depth** of 10 in a QoS profile:

```
<history>
   <kind>KEEP_LAST_HISTORY_QOS</kind>
   <depth>10</depth>
</history>
```

You can see this in the **News** example (provided for C, C++, Java, and Ada) in the file **USER_QOS_PROFILES.xml**.

Lifespan Duration:

The Lifespan QoS policy contains a field **duration** that indicates for how long each DDS sample remains valid. The duration is measured relative to the DDS sample's *reception timestamp*, which means that it doesn't include the latency of the underlying transport.

To specify a Lifespan duration of six seconds:

```
<lifespan>
   <duration>
       <sec>6</sec>
       <nanosec>0</nanosec>
   </duration>
</lifespan>
```

You can see this in the **News** example (provided for C, C++, Java, and Ada) in the file **USER_QOS_PROFILES.xml**.

6.2.2.2 Running & Verifying

The **News** example never *takes* DDS samples from the middleware, it only *reads* DDS samples, so DDS data samples that have already been viewed by the subscribing application remain in the middleware's internal cache until they are expired by their history depth or lifespan duration contract. These previously viewed DDS samples are displayed by the subscribing application as "cached" to make them easy to spot. See how "CNN" articles are expired:

Figure 6.3: Using History and Lifespan QoS

```
Available articles as of 10:02:48 AM:
    [10:02:47 AM] From Reuters    : "lorem ipsum 1"
    [10:02:47 AM] From CNN        : "lorem ipsum 1"
    [10:02:47 AM] From Reuters    : "lorem ipsum 2"
    [10:02:47 AM] From Reuters    : "lorem ipsum 3"
    [10:02:47 AM] From Reuters    : "lorem ipsum 4"
    [10:02:47 AM] From CNN        : "lorem ipsum 2"
    [10:02:48 AM] From Reuters    : "lorem ipsum 5"
Available articles as of 10:02:50 AM:
    [10:02:47 AM] From CNN        : "lorem ipsum 1" (cached)
    [10:02:47 AM] From CNN        : "lorem ipsum 2" (cached)
    [10:02:48 AM] From Reuters    : "lorem ipsum 6"
    [10:02:48 AM] From Reuters    : "lorem ipsum 7"
    [10:02:48 AM] From CNN        : "lorem ipsum 3"
    [10:02:48 AM] From Reuters    : "lorem ipsum 8"
    [10:02:48 AM] From Reuters    : "lorem ipsum 9"
    [10:02:49 AM] From Reuters    : "lorem ipsum 10"
    [10:02:49 AM] From CNN        : "lorem ipsum 4"
    [10:02:49 AM] From Reuters    : "lorem ipsum 11"
    [10:02:49 AM] From Reuters    : "lorem ipsum 12"
    [10:02:49 AM] From Reuters    : "lorem ipsum 13"
    [10:02:49 AM] From CNN        : "lorem ipsum 5"
    [10:02:49 AM] From Reuters    : "lorem ipsum 14"
    [10:02:50 AM] From Reuters    : "lorem ipsum 15"
Available articles as of 10:02:52 AM:
    [10:02:47 AM] From CNN        : "lorem ipsum 1" (cached)
    [10:02:47 AM] From CNN        : "lorem ipsum 2" (cached)
    [10:02:48 AM] From CNN        : "lorem ipsum 3" (cached)
    [10:02:49 AM] From CNN        : "lorem ipsum 4" (cached)
    [10:02:49 AM] From CNN        : "lorem ipsum 5" (cached)
    [10:02:50 AM] From Reuters    : "lorem ipsum 16"
    [10:02:50 AM] From CNN        : "lorem ipsum 6"
    [10:02:50 AM] From Reuters    : "lorem ipsum 17"
    [10:02:50 AM] From Reuters    : "lorem ipsum 18"
    [10:02:50 AM] From Reuters    : "lorem ipsum 19"
    [10:02:50 AM] From CNN        : "lorem ipsum 7"
    [10:02:51 AM] From Reuters    : "lorem ipsum 20"
    [10:02:51 AM] From Reuters    : "lorem ipsum 21"
    [10:02:51 AM] From Reuters    : "lorem ipsum 22"
    [10:02:51 AM] From CNN        : "lorem ipsum 8"
    [10:02:51 AM] From Reuters    : "lorem ipsum 23"
    [10:02:51 AM] From Reuters    : "lorem ipsum 24"
    [10:02:52 AM] From Reuters    : "lorem ipsum 25"
    [10:02:52 AM] From CNN        : "lorem ipsum 9"
Available articles as of 10:02:54 AM:
    [10:02:48 AM] From CNN        : "lorem ipsum 3" (cached)
    [10:02:49 AM] From CNN        : "lorem ipsum 4" (cached)
    [10:02:49 AM] From CNN        : "lorem ipsum 5" (cached)
    [10:02:50 AM] From CNN        : "lorem ipsum 6" (cached)
    [10:02:50 AM] From CNN        : "lorem ipsum 7" (cached)
    [10:02:51 AM] From CNN        : "lorem ipsum 8" (cached)
    [10:02:52 AM] From CNN        : "lorem ipsum 9" (cached)
```

Article 1 initially received from Reuters and CNN

CNN articles still available

Reuters articles 1-5 expired due to history depth (10 samples)

CNN articles 1 & 2 expired due to lifespan (6 seconds)

Remember that there is a strong analogy between DDS and relational databases: the key of a Topic is like the primary key of a database table, and the instance corresponding to that key is like the table row corresponding to that primary key.

It's important to understand that the data type used by this example is keyed on the name of the news outlet and that the history depth is enforced on a per-instance basis. You can see the effect in Figure 6.3: Using History and Lifespan QoS on the previous page: even though Reuters publishes articles faster than CNN, the Reuters articles do not "starve out" the CNN articles; each outlet gets its own depth DDS samples. (Figure 6.3: Using History and Lifespan QoS on the previous page only shows articles from CNN and Reuters, because that makes it easier to fit more data on the page. If you run the example without a content filter, you will see the same effect across all news outlets.).

Next, we will change the history **depth** and lifespan **duration** in the file **USER_QOS_PROFILES.xml** to see how the example's output changes.

Set the history **depth** to **1.**

Run the example again with the content-based filter, shown here in the Java example:

```
> ./run.sh sub -f "key='CNN' OR key='Reuters'"
```

You will only see articles from CNN and Reuters, and only the last value published for each (as of the end of the two-second polling period).

Now set the history **depth** to 10 and decrease the **lifespan** to three seconds.

With these settings and at the rates used by this example, the lifespan will always take effect before the history depth. Run the example again, this time without a content filter. Notice how the subscribing application sees all of the data that was published during the last two-second period as well as the data that was published in the latter half of the previous period.

Reduce the **lifespan** again, this time to **1** second.

If you run the example now, you will see no cached articles at all. Do you understand why? The subscribing application's polling period is two seconds, so by the time the next poll comes, everything seen the previous time has expired.

Try changing the history **depth** and lifespan duration in the file **USER_QOS_PROFILES.xml** to see how the example's output changes.

6.2.3 Time-Based Filtering

A time-based filter allows you to specify a *minimum separation* between the DDS data DDS samples your subscribing application receives. If data is published faster than this rate, the middleware will discard the intervening DDS samples.

Such a filter is most often used to down-sample high-rate periodic data (see also 6.5 Receiving Notifications When Data Delivery Is Late on page 75) but it can also be used to limit data rates for aperiodic-but-bursty data streams. Time-based filters have several applications:

- You can limit data update rates for applications in which rapid updates would be unnecessary or inappropriate. For example, a graphical user interface should typically not update itself more than a few times each second; more frequent updates can cause flickering or make it difficult for a human operator to perceive the correct values.
- You can reduce the CPU requirements for less-capable subscribing machines to improve their performance. If your data stream is reliable, helping slow readers keep up can actually improve the effective throughput for all readers by preventing them from throttling the writer.
- In some cases, you can reduce your network bandwidth utilization, because the writer can transparently discard unwanted data before it is even sent on the network.

6.2.3.1 Implementation

Time-based filters are specified using the TimeBasedFilter QoS policy. It only applies to *DataReaders*.

For more information about this QoS policy, consult the API Reference HTML documentation. Open **ReadMe.html** and select a programming language, then select **Modules, RTI Connext DDS API Reference, Infrastructure, QoS Policies**. For Ada: open **<NDDSHOME>/doc/api/connext_dds/api_ada/index.html** and select **Infrastructure Module, DDSQosTypesModule**.

You can specify the QoS policies either in your application code or in one or more XML files. Both mechanisms are functionally equivalent; the **News** example uses the XML mechanism. For more information, see the section on “Configuring QoS with XML” in the *RTI Connext DDS Core Libraries User's Manual*.

To specify a time-based filter in a QoS policy:

```
<!--
<time_based_filter>
   <minimum_separation>
       <sec>1</sec>
       <nanosec>1</nanosec>
   </minimum_separation>
</time_based_filter>
<deadline>
   <period>
       <sec>3</sec>
       <nanosec>0</nanosec>
   </period>
</deadline>
-->
```

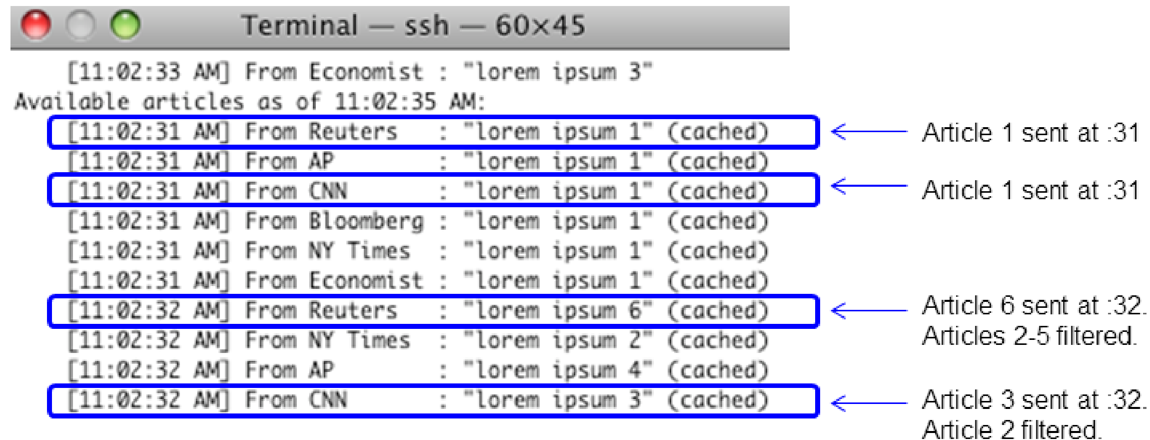
You can see this in the file **USER_QOS_PROFILES.xml** provided with the C, C++, Java, and Ada **News** example (uncomment it to specify a time-based filter).

At the same time we implement a time-based filter in the News example, we increase the deadline period. See the accompanying comment in the XML file as well as the API Reference HTML documentation for more information about using the Deadline and TimeBasedFilter QoS policies together.

6.2.3.2 Running & Verifying

Figure 6.4: Using a Time-Based Filter below shows some of the output after activating the filter:

Figure 6.4: Using a Time-Based Filter



Because you set the time-based filter to one second, you will not see more than two updates for any single news outlet in any given period, because the period is two seconds long.

6.3 Accessing Historical Data when Joining the Network

In 6.2 Subscribing Only to Relevant Data on page 60, you learned how to specify which data on the network is of interest to your application. The same QoS parameters can also apply to late-joining subscribers, applications that subscribe to a topic after some amount of data has already been published on that topic. This concept—storing sent data within the middleware—is referred to as *durability*. You only need to indicate the degree of durability in which you're interested:

- **Volatile**. Data is relevant to current subscribers only. Once it has been acknowledged (if reliable communication has been turned on), it can be removed from the service. This level of durability is the default; if you specify nothing, you will get this behavior.
- **Transient local**. Data that has been sent may be relevant to late-joining subscribers (subject to any history depth, lifespan, and content- and/or time-based filters defined). Historical data will be cached with the *DataWriter* that originally produced it. Once that writer has been shut down for any reason, intentionally or unintentionally, however, the data will no longer be available. This lightweight level of durability is appropriate for non-critical data streams without stringent data availability requirements.
- **Transient**. Data that has been sent may be relevant to late-joining subscribers and will be stored externally to the *DataWriter* that produced that data. This level of durability requires one or more instances of *RTI Persistence Service* on your network. As long as one or more of these persistence

service instances is functional, the durable data will continue to be available, even if the original *DataWriter* shuts down or fails. However, if all instances of the persistence service shut down or fail, the durable data they were maintaining will be lost. This level of durability provides a higher level of fault tolerance and availability than does transient-local durability without the performance or management overhead of a database.

- **Persistent**. Data that has been sent may be relevant to late-joining subscribers and will be stored externally to the *DataWriter* that produced that data in a relational database. This level of durability requires one or more instances of *RTI Persistence Service* on your network. It provides the greatest degree of assurance for your most critical data streams, because even if all data writers and all persistence server instances fail, your data can nevertheless be restored and made available to subscribers when you restart the persistence service.

As you can see, the level of durability indicates not only *whether* historical data will be made available to late-joining subscribers; it also indicates the level of fault tolerance with which that data will be made available. Learn more about durability, including the *RTI Persistence Service*, by reading the section on "Mechanisms for Achieving Information Durability and Persistence" in the *RTI Connext DDS Core Libraries User's Manual*.

6.3.1 Implementation

To configure data durability, use the Durability QoS policy on your *DataReader* and/or *DataWriter*. The degrees of durability described in 6.3 Accessing Historical Data when Joining the Network on the previous page are represented as an enumerated durability kind.

For more information about this QoS policy, consult the API Reference HTML documentation. Open **ReadMe.html** and select the API documentation for your language then select **Modules, RTI Connext DDS API Reference, Infrastructure, QoS Policies**. For Ada: open **<NDDSHOME>/doc/api/connext_dds/api_ada/index.html and select Infrastructure Module, DDSQosTypesModule**.

You can specify the QoS policies either in your application code or in one or more XML files. Both mechanisms are functionally equivalent; the News example uses the XML mechanism. For more information, see the chapter on "Configuring QoS with XML" in the *RTI Connext DDS Core Libraries User's Manual*.

Here is an example of how to configure the Durability QoS policy in a QoS profile:

```
<durability>
       <kind>TRANSIENT_LOCAL_DURABILITY_QOS</kind>
</durability>
```

You can see this in the **news** example (provided for C, C++, Java, and Ada) in the file **USER_QOS_PROFILES.xml**.

The above configuration indicates that the *DataWriter* should maintain data it has published on behalf of later-joining *DataReaders*, and that *DataReaders* should expect to receive historical data when they join

the network. However, if a *DataWriter* starts up, publishes some data, and then shuts down, a *DataReader* that subsequently starts up will not receive that data.

Durability, like some other QoS policies, has *request-offer* semantics: the *DataWriter* must offer a level of service that is greater than or equal to the level of service requested by the *DataReader*. For example, a *DataReader* may request only volatile durability, while the *DataWriter* may offer transient durability. In such a case, the two will be able to communicate. However, if the situation were reversed, they would not be able to communicate.

6.3.2 Running & Verifying

Run the **NewsPublisher** and wait several seconds. Then start the **NewsSubscriber**. Look at the time stamps printed next to the received data: you will see that the subscribing application receives data that was published before it started up.

Now do the same thing again, but first modify the configuration file by commenting the durability configuration. You will see that the subscribing application does not receive any data that was sent prior to when it joined the network.

6.4 Caching Data within the Middleware

When you receive data from the middleware in your subscribing application, you may be able to process all of the data immediately and then discard it. Frequently, however, you will need to store it somewhere in order to process it later. Since you've already expressed to the middleware how long your data remains relevant (see 6.2 Subscribing Only to Relevant Data on page 60), wouldn't it be nice if you could take advantage of the middleware's own data cache rather than implementing your own? You can.

When a *DataReader* reads data from the network, it places the DDS samples, in order, into its internal cache. When you're ready to view that data (either because you received a notification that it was available or because you decided to poll), you use one of two families of methods:

- **take**: Read the data from the cache and simultaneously *remove it from that cache*. Future access to that *DataReader's* cache will not see any data that was previously taken from the cache. This behavior is similar to the behavior of "receive" methods provided by traditional messaging middleware implementations. The generated Ada example uses this method.
- **read**: Read the data from the cache but *leave it in the cache* so that it can potentially be read again (subject to any lifespan or history depth you may have specified). The **news** example for C, C++ and Java uses this method.

When you read or take data from a *DataReader*, you can indicate that you wish to access all of the data in the cache, all up to a certain maximum number of DDS samples, all of the new DDS samples that you have never read before, and/or various other qualifiers. If your topic is keyed, you can choose to access the DDS samples of all instances at once, or you can read/take one instance at a time. For more information

about keys and instances, see Section 2.2.2, "DDS Samples, Instances, and Keys" in the *RTI Connext DDS Core Libraries User's Manual*.

6.4.1 Implementation

The call to **read** looks like this:

In C++:

```
DDS_ReturnCode_t result = _reader->read(
       articles, // fill in data here
       articleInfos, // fill in parallel meta-data here
       DDS_LENGTH_UNLIMITED, // any # articles
       DDS_ANY_SAMPLE_STATE, DDS_ANY_VIEW_STATE, DDS_ANY_INSTANCE_STATE);
if (result == DDS_RETCODE_NO_DATA) {
       // nothing to read; go back to sleep
}
if (result != DDS_RETCODE_OK) {
       // an error occurred: stop reading
       throw std::runtime_error("A read error occurred: " + result);
}
// Process data...
_reader->return_loan(articles, articleInfos);
```

In Java:

```
try {
       _reader.read(
               articles,       // fill in data here
               articleInfos,   // fill in parallel meta-data here
               ResourceLimitsQosPolicy.LENGTH_UNLIMITED, // any # articles
               SampleStateKind.ANY_SAMPLE_STATE,
               ViewStateKind.ANY_VIEW_STATE,
               InstanceStateKind.ANY_INSTANCE_STATE);
       // Process data...
}  catch (RETCODE_NO_DATA noData) {
       // nothing to read; go back to sleep
}  catch (RETCODE_ERROR ex) {
       // an error occurred: stop reading
       throw ex;
} finally {
       _reader.return_loan(articles, articleInfos);
}
```

In Ada:

```
declare
received_data : aliased DDS.KeyedString_Seq.Sequence;
sample_info : aliased DDS.SampleInfo_Seq.Sequence;
begin
       reader.Read(
               received_data'Access, sample_info'Access,
```

```
                    DDS.LENGTH_UNLIMITED,
                    DDS.ANY_SAMPLE_STATE,
                    DDS.ANY_VIEW_STATE,
                    DDS.ANY_INSTANCE_STATE);
        for i in 1 .. DDS.KeyedString_Seq.Get_Length(received_data'Access)
        loop
                printArticle(
                    DDS.KeyedString_Seq.Get (received_data'Access, i),
                    DDS.SampleInfo_Seq.Get (sample_info'Access, i));
        end loop;
        reader.Return_Loan (received_data'Access, sample_info'Access);
        exception
                when DDS.NO_DATA =>
                    null; -- ignore this error
                    -- nothing to read; go back to sleep
                when DDS.ERROR =>
                    -- an error occurred: stop reading
                    raise;
        end;
end;
```

The **read** method takes several arguments:

- Data and SampleInfo sequences:

  The first two arguments to **read** or **take** are lists: for the DDS samples themselves and, in parallel, for meta-data about those DDS samples. In most cases, you will pass these collections into the middleware empty, and the middleware will fill them for you. The objects it places into these sequences are loaned directly from the *DataReader's* cache in order to avoid unnecessary object allocations or copies. When you are done with them, call **return_loan.**

  If you would rather read the DDS samples into your own memory instead of taking a loan from the middleware, simple pass in sequences in which the contents have already been deeply allocated. The *DataReader* will copy over the state of the objects you provide to it instead of loaning you its own objects. In this case, you do not need to call **return_loan**.

- Number of DDS samples to read:

  If you are only prepared to read a certain number of DDS samples at a time, you can indicate that to the *DataReader*. In most cases, you will probably just use the constant **LENGTH_UNLIMITED**, which indicates that you are prepared to handle as many DDS samples as the *DataReader* has available.

- Sample state:

  The sample state indicates whether or not an individual DDS sample has been observed by a previous call to **read**. (The **news** example uses this state to decide whether or not to append "(cached)"

to the data it prints out.) By passing a sample state mask to the *DataReader*, you indicate whether you are interested in all DDS samples, only those DDS samples you've never seen before, or only those DDS samples you have seen before. The most common value passed here is **ANY_SAMPLE_STATE**.

- View state:

  The view state indicates whether the instance to which a DDS sample belongs has been observed by a previous call to **read** or **take**. By passing a view state mask to the *DataReader*, you can indicate whether you're interested in all instances, only those instances that you have never seen before (**NEW** instances), or only those instances that you *have* seen before (**NOT_NEW** instances). The most common value passed here is **ANY_VIEW_STATE**.

- Instance state:

  The instance state indicates whether the instance to which a DDS sample belongs is still *alive* (**ALIVE**), whether it has been disposed (**NOT_ALIVE_DISPOSED**), or whether the *DataWriter* has simply gone away or stopped writing it without disposing it (**NOT_ALIVE_NO_WRITERS**). The most common value passed here is **ANY_INSTANCE_STATE**. For more information about the data lifecycle, consult the *RTI Connext DDS Core Libraries User's Manual* and the API Reference HTML documentation.

  Unlike reading from a socket directly, or calling **receive** in JMS, a **read** or **take** is non-blocking: the call returns immediately. If no data was available to be read, it will return (in C and C++) or throw (in Java, .NET[a], and Ada) a **NO_DATA** result.

  Connext DDS also offers additional variations on the **read()** and **take()** methods to allow you to view different "slices" of your data at a time:

  - You can view one instance at a time with **read_instance()**, **take_instance()**, **read_next_instance()**, and **take_next_instance()**.
  - You can view a single DDS sample at a time with **read_next_sample()** and **take_next_sample()**.

For more information about these and other variations, see the API Reference HTML documentation: open **ReadMe.html**, select a language, and look under **Modules, Subscription Module, DataReader Support, FooDataReader.** For Ada, open **<NDDSHOME>/doc/api/connext_dds/api_ada/index.html** and look under **Subscription Module, DDSReaderModule, DDS.Typed_DataReader_Generic**.

[a]*Connext DDS* .NET language binding is currently supported for C# and C++/CLI.

6.4.2 Running & Verifying

To see the difference between **read()** and **take()** semantics, replace “read” with “take” (the arguments are the same), rebuild the example, and run again. You will see that “(cached)” is never printed, because every DDS sample will be removed from the cache as soon as it is viewed for the first time.

6.5 Receiving Notifications When Data Delivery Is Late

Many applications expect data to be sent and received periodically (or quasi-periodically). They typically expect at least one DDS data sample to arrive during each period; a failure of data to arrive may or may not indicate a serious problem, but is probably something about which the application would like to receive notifications. For example:

- A vehicle may report its current position to its home base every second.
- Each sensor in a sensor network reports a new reading every 0.5 seconds.
- A radar reports the position of each object that it’s tracking every 0.2 seconds.

If any vehicle, any sensor, or any radar track fails to yield an update within its promised period, another part of the system, or perhaps its human operator, may need to take a corrective action.

(In addition to built-in deadline support, *Connext DDS* has other features useful to applications that publish and/or receive data periodically. For example, it’s possible to down-sample high-rate periodic data; see 6.2 Subscribing Only to Relevant Data on page 60.

6.5.1 Implementation

Deadline enforcement is comprised of two parts: (1) QoS policies that specify the deadline contracts and (2) listener callbacks that are notified if those contracts are violated.

Deadlines are enforced independently for *DataWriters* and *DataReaders*. However, the Deadline QoS policy, like some other policies, has request-offer semantics: the *DataWriter* must offer a level of service that is the same or better than the level of service the *DataReader* requests. For example, if a *DataWriter* promises to publish data at least once every second, it will be able to communicate with a *DataReader* that expects to receive data at least once every two seconds. However, if the *DataReader* declares that it expects data twice a second, and the *DataWriter* only promises to publish updates only once a second, they will not be able to communicate.

6.5.1.1 Offered Deadlines

A *DataWriter* promises to publish data at a certain rate by providing a finite value for the Deadline QoS policy, either in source code or in one or more XML configuration files. (Both mechanisms are functionally equivalent; the **News** example in C, C++, Java, and Ada uses XML files. For more information about this mechanism, see the section on “Configuring QoS with XML” in the *RTI Connext DDS Core Libraries User's Manual.*)

The file **USER_QOS_PROFILES.xml** in the **News** example for C, C++, Java, and Ada contains the following Deadline QoS policy configuration, which applies to both *DataWriters* and **DataReaders**:

```
<deadline>
   <period>
       <sec>2</sec>
       <nanosec>0</nanosec>
   </period>
</deadline>
```

The *DataWriter* thus promises to publish at least one DDS data sample—of *each instance*—every two seconds.

If a period of two seconds elapses from the time the *DataWriter* last sent a DDS sample of a particular instance, the writer's listener—if one is installed—will receive a callback to its **on_offered_deadline_missed** method. The **News** example does not actually install a *DataWriterListener*. See the section on requested deadlines below; the *DataWriterListener* works in a way that's parallel to the *DataReaderListener*.

6.5.1.2 Requested Deadlines

The *DataReader* declares that it expects to receive at least one DDS data sample of each instance within a given period using the Deadline QoS policy. See the example XML configuration above.

If the declared deadline period elapses since the *DataReader* received the last DDS sample of some instance, that reader's listener—if any—will be invoked. The listener's **on_requested_deadline_missed()** will receive a call informing the application of the missed deadline.

To install a *DataReaderListener:*

In C++:

```
DDSDataReader* reader = participant->create_datareader(
   topic,
   DDS_DATAREADER_QOS_DEFAULT,
   &_listener, // listener
   DDS_STATUS_MASK_ALL); // all callbacks
if (reader == NULL) {
   throw std::runtime_error("Unable to create DataReader");
}
```

In Java:

```
DataReader reader = participant.create_datareader(
   topic,
   Subscriber.DATAREADER_QOS_DEFAULT,
   new ArticleDeliveryStatusListener(), // listener
```

```
   StatusKind.STATUS_MASK_ALL); // all callbacks
if (reader == null) {
   throw new IllegalStateException("Unable to create DataReader");
}
```

In Ada:

```
declare
    readerListener : ArticleDeliveryStatusListener;
begin
   reader := participant.Create_DataReader
               (topic.As_TopicDescription,
                DDS.Subscriber.DATAREADER_QOS_DEFAULT,
                readerListener'Unchecked_Access, -- listener
                DDS.STATUS_MASK_ALL); -- all callbacks
    if reader = null then
        Put_Line (Standard_Error, "Unable to create DataReader");
        return;
    end if;
end;
```

There are two listener-related arguments to provide:

- **Listener**: The listener object itself, which must implement some subset of the callbacks defined by the *DataReaderListener* supertype.
- **Listener mask**: Which of the callbacks you'd like to receive. In most cases, you will use one of the constants **STATUS_MASK_ALL** (if you are providing a non-null listener object) of **STATUS_MASK_NONE** (if you are not providing a listener). There are some cases in which you might want to specify a different listener mask; see the *RTI Connext DDS Core Libraries User's Manual* and API Reference HTML documentation for more information.

Let's look at a very simple implementation of the **on_requested_deadline_missed** callback that prints the value of the key (i.e., the news outlet name) for the instance whose deadline was missed. You can see this in the **News** example provided with C, C++, and Java.

In C++:

```
void ArticleDeliveryStatusListener::on_requested_deadline_missed(
    DDSDataReader* reader,
    const DDS_RequestedDeadlineMissedStatus& status) {
        DDS_KeyedString keyHolder;
        DDSKeyedStringDataReader* typedReader =
            DDSKeyedStringDataReader::narrow(reader);
        typedReader->get_key_value(keyHolder,
            status.last_instance_handle);
        std::cout << "->Callback: requested deadline missed: "
            << keyHolder.key
```

```
            << std::endl;
}
```

In Java:

```
public void on_requested_deadline_missed(
    DataReader reader,
    RequestedDeadlineMissedStatus status) {
    KeyedString keyHolder = new KeyedString();
    reader.get_key_value_untyped(keyHolder,
        status.last_instance_handle);
    System.out.println("->Callback: requested deadline missed: " +
        keyHolder.key);
}
```

In Ada:

```
procedure On_Requested_Deadline_Missed
    (Self : not null access Ref;
    The_Reader : in DDS.DataReaderListener.DataReader_Access;
    Status : in DDS.RequestedDeadlineMissedStatus)
is
    pragma Unreferenced (Self);
    pragma Unreferenced (The_Reader);
    pragma Unreferenced (Status);
begin
    Put_Line ("->Callback: requested deadline missed.");
end On_Requested_Deadline_Missed;
```

6.5.2 Running & Verifying

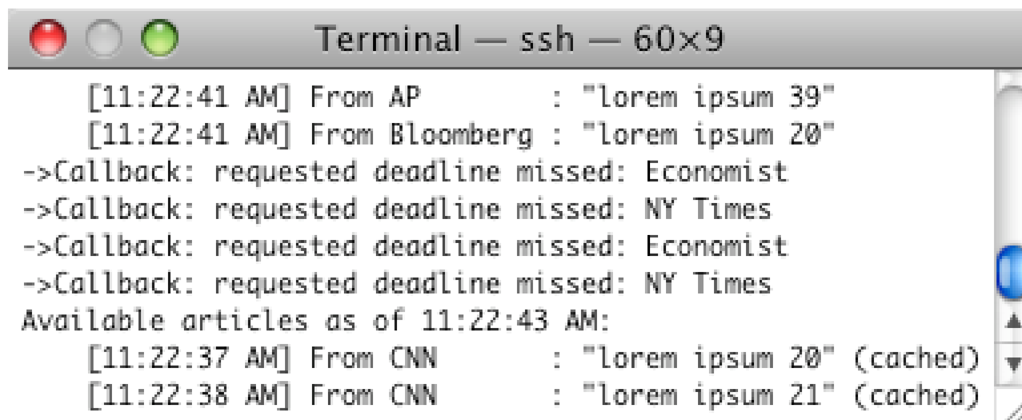
Modify the file **USER_QOS_PROFILES.xml** to decrease the deadline to one second:

```
<deadline>
   <period>
        <sec>1</sec>
        <nanosec>0</nanosec>
    </period>
</deadline>
```

Note that, if you have a *DataReader*- or *DataWriter*-level deadline specified (inside the file's **datareader_qos** or **datawriter_qos** elements, respectively)—possibly because you previously modified the configuration in 6.2.3 Time-Based Filtering on page 67—it is overriding the topic-level configuration. Be careful that you don't modify a deadline specification that will only be overridden later and not take effect.

You will see output similar to Figure 6.5: Using a Shorter Deadline on the next page:

Figure 6.5: Using a Shorter Deadline



Chapter 7 Design Patterns for High Performance

In this section, you will learn how to implement some common performance-oriented design patterns. As you have learned, one of the advantages to using *Connext DDS* is that you can easily tune your application without changing its code, simply by updating the XML-based Quality of Service (QoS) parameters.

We will build on the examples used in 4.1 Building and Running "Hello, World" on page 14 to demonstrate the different use-cases. The example applications (**Hello_builtin, Hello_idl**, and **Hello_dynamic**[a]), provide the same functionality but use different data types in order to help you understand the different type-definition mechanisms offered by *Connext DDS* and their tradeoffs. They implement a simple throughput test: the publisher sends a payload to the subscriber, which periodically prints out some basic statistics. You can use this simple test to quickly see the effects of your system design choices: programming language, target machines, QoS configurations, and so on.

The QoS parameters do not depend on the language used for your application, and seldom on the operating system (there are few key exceptions), so you should be able to use the XML files with the example in the language of your choice.

This section includes:

- 7.1 Building and Running the Code Examples on the next page
- 7.2 Reliable Messaging on page 84
- 7.3 High Throughput for Streaming Data on page 88
- 7.4 Sending Large Data on page 91
- 7.5 Streaming Data over Unreliable Network Connections on page 95

---

[a]Dynamic DDS types are not supported when using ***Ada Language Support.***

7.1 Building and Running the Code Examples

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

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

Default path to the examples:

- macOS systems: **/Users/<*your user name*>/rti_workspace/6.0.1/examples**
- Linux systems: **/home/<*your user name*>/rti_workspace/6.0.1/examples**
- Windows systems: ***<your Windows documents folder>*\rti_workspace\6.0.1\examples**

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

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

You can find the source for the **Hello_builtin** example for Java in **<path to examples>/connext_dds/-java/hello_builtin**; the equivalent source code in other supported languages is in the directories **<path to examples>/connext_dds/<language>**. The **hello_idl** and **hello_dynamic** examples are in parallel directories under **<path to examples>/connext_dds/<language>**.

The examples perform these steps:

1. Parse their command-line arguments.
2. Check if the Quality of Service (QoS) file can be located.

   The XML file that sets the Quality of Service (QoS) parameters is either loaded from the file **USER_QOS_PROFILES**.xml in the current directory of the program, or from the environment variable **NDDS_QOS_PROFILES**. For more information on how to use QoS profiles, see the section on "Configuring QoS with XML" in the *RTI Connext DDS Core Libraries User's Manual.*
3. On the publishing side, send text strings as fast as possible, prefixing each one with a serial number.
4. On the subscribing side, receive these strings, keeping track of the highest number seen, as well as other statistics. Print these out periodically.

Figure 7.1: Example Output from Subscribing Applications in C, C++, or Java

```
Terminal — ssh — 75×25
> ./objs/i86Linux2.6gcc3.4.3/Hello sub --domain 13
Hello Example Application
Copyright 2008 Real-Time Innovations, Inc.


Sec.from|Total     |Total Lost|Curr Lost |Average   |Current   |Throughput
start   |samples   |samples   |samples   |smpls/sec |smpls/sec |Mbps
--------+----------+----------+----------+----------+----------+----------
       1       3984          0          0    3984.00    3984.00      31.09
       2       8112          0          0    4056.00    4128.00      32.22
       3      12096          0          0    4032.00    3984.00      31.09
       4      16128          0          0    4032.00    4032.00      31.47
       5      20112          0          0    4022.40    3984.00      31.09
       6      24144          0          0    4024.00    4032.00      31.47
       7      28176          0          0    4025.14    4032.00      31.47
       8      32160          0          0    4020.00    3984.00      31.09
       9      36432          0          0    4048.00    4272.00      33.34
      10      40800          0          0    4080.00    4368.00      34.09
      11      45168          0          0    4106.18    4368.00      34.09
      12      49536          0          0    4128.00    4368.00      34.09
      13      53904          0          0    4146.46    4368.00      34.09
      14      58272          0          0    4162.29    4368.00      34.09
      15      62304          0          0    4153.60    4032.00      31.47
      16      66288          0          0    4143.00    3984.00      31.09
      17      70320          0          0    4136.47    4032.00      31.47
```

The steps for compiling and running the program are the same as mentioned in 4.1 Building and Running "Hello, World" on page 14.

Run the publisher and subscriber from the **<path to examples>/connext_dds/<language>/hello_builtin** directory using one of the QoS profile files provided in **examples/connext_dds/qos** by copying it into your current working directory with the file name **USER_QOS_PROFILES.xml**. You should see output like the following from the subscribing application:

7.1.1 Understanding the Performance Results

You will see several columns in the subscriber-side output, similar to Figure 7.1: Example Output from Subscribing Applications in C, C++, or Java above.

- **Seconds from start:**
  The number of seconds the subscribing application has been running. It will print out one line per second.

- **Total DDS samples:**
  The number of DDS data samples that the subscribing application has received since it started running.

- **Total lost DDS samples:**
The number of DDS samples that were lost in transit and could not be re-paired, since the subscribing application started running. If you are using a QoS profile configured for strict reliability, you can expect this column to always display 0. If you are running in best-effort mode, or in a limited-reliability mode (e.g., you have configured your History QoS policy to only keep the most recent DDS sample), you may see non-zero values here.

- **Current lost DDS samples:**
The number of DDS samples that were lost in transit and could not be repaired, since the last status line was printed. See the description of the previous column for information about what values to expect here.

- **Average DDS samples per second:**
The mean number of DDS data samples received per second since the subscribing application started running. By default, these example applications send DDS data samples that are 1 KB in size. (You can change this by passing a different size, in bytes, to the publishing application with **--size**.)

- **Current DDS samples per second:**
The number of DDS data samples received since the subscribing application printed the last status line.

- **Throughput megabits per second:**

  The throughput from the publishing application to the subscribing application, in bits per second, since the subscribing application printed the previous status line. The value in this column is equivalent to the current DDS samples per second multiplied by the number of bits in an individual DDS sample.

  With small sample sizes, the fixed "cost" of traversing your operating system's network stack is greater than the cost of actually transmitting the data; as you increase the sample size, you will see the throughput more closely approach the theoretical throughput of your network.

  By batching multiple DDS data samples into a single network packet, as the high-throughput example QoS profile does, you should be able to saturate a gigabit Ethernet network with sample sizes as small as 100-200 bytes. Without batching DDS samples, you should be able to saturate the network with DDS samples of a few kilobytes. The difference is due to the performance limitations of the network transport; enterprise-class platforms with commodity Ethernet interfaces can typically execute tens of thousands of **send()'**s per second. In contrast, saturating a high-throughput network link with data sizes of less than a kilobyte requires hundreds of thousands of DDS samples per second.

7.1.2 Is this the Best Possible Performance?

The performance of an application depends heavily on the operating system, the network, and how it configures and uses the middleware. This example is just a starting point; tuning is important for a production application. RTI can help you get the most out of your platform and the middleware.

To get a sense for how an application's behavior changes with different QoS contracts, try the other provided example QoS profiles and see how the printed results change.

7.2 Reliable Messaging

Packets sent by a middleware may be lost by the physical network or dropped by routers, switches and even the operating system of the subscribing applications when buffers become full. In reliable messaging, the middleware keeps track of whether or not data sent has been received by subscribing applications, and will resend data that was lost on transmission.

Like most reliable protocols (including TCP), the reliability protocol used by RTI uses additional packets on the network, called metadata, to know when user data packets are lost and need to be resent. RTI offers the user a comprehensive set of tunable parameters that control how many and how often metadata packets are sent, how much memory is used for internal buffers that help overcome intermittent data losses, and how to detect and respond to a reliable subscriber that either falls behind or otherwise disconnects.

When users want applications to exchange messages reliably, there is always a need to trade-off between performance and memory requirements. When strictly reliable communication is enabled, every written DDS sample will be kept by *Connext DDS* inside an internal buffer until all known reliable subscribers acknowledge receiving the DDS sample[a].

If the publisher writes DDS samples faster than subscribers can acknowledge receiving, this internal buffer will eventually be completely filled, exhausting all the available space—in that case, further writes by the publishing application will block. Similarly, on the subscriber side, when a DDS sample is received, it is stored inside an internal receive buffer, waiting for the application to take the data for processing. If the subscribing application doesn't take the received DDS samples fast enough, the internal receive buffer may fill up—in that case, newly received data will be discarded and would need to be repaired by the reliable protocol.

Although the size of those buffers can be controlled from the QoS, you can also use QoS to control what *Connext DDS* will do when the space available in one of those buffers is exhausted. There are two possible scenarios for both the publisher and subscriber:

---

[a] *Connext DDS* also supports reliability based only on negative acknowledgements ("NACK-only reliability"). This feature is described in detail in the *User's Manual* (Section 6.5.2.3) but is beyond the scope of this document.

- **Publishing side:** If **write()** is called and there is no more room in the *DataWriter's* buffer, *Connext DDS* can:
    1. Temporarily block the write operation until there is room on this buffer (for example, when one or more DDS samples is acknowledged to have been received from all the subscribers).
    2. Drop the oldest DDS sample from the queue to make room for the new one.
- **Subscribing side:** If a DDS sample is received (from a publisher) and there is no more room on the *DataReader's* buffer:
    1. Drop the DDS sample as if it was never received. The subscribing application will send a negative acknowledgement requesting that the DDS sample be resent.
    2. Drop the oldest DDS sample from the queue to make room for the new one.

7.2.1 Implementation

There are many variables to consider, and finding the optimum values to the queue size and the right policy for the buffers depends on the type of data being exchanged, the rate of which the data is written, the nature of the communication between nodes and various other factors.

The *RTI Connext DDS Core Libraries User's Manual* dedicates an entire section to the reliability protocol, providing details on choosing the correct values for the QoS based on the system configuration. For more information, refer to Chapter 10 in the *User's Manual*.

The following sections highlight the key QoS settings needed to achieve strict reliability. In the **reliable.xml** QoS profile file, you will find many other settings besides the ones described here. A detailed description of these QoS is outside the scope of this document, and for further information, refers to the comments in the QoS profile and in the *User's Manual*.

7.2.1.1 Enable Reliable Communication

The QoS that control the kind of communication is the Reliability QoS of the *DataWriter* and *DataReader*:

```
<datawriter_qos>
   ...
   <reliability>
       <kind>RELIABLE_RELIABILITY_QOS</kind>
       <max_blocking_time>
           <sec>5</sec>
           <nanosec>0</nanosec>
       </max_blocking_time>
    </reliability>
    ...
</datawriter_qos>
...
<datareader_qos>
    <reliability>
        <kind>RELIABLE_RELIABILITY_QOS</kind>
    </reliability>
</datareader_qos>
```

This section of the QoS file enables reliability on the *DataReader* and *DataWriter*, and tells the middleware that a call to **write()** may block up to 5 seconds if the *DataWriter's* cache is full of unacknowledged DDS samples. If no space opens up in 5 seconds, **write()** will return with a timeout indicating that the write operation failed and that the data was not sent.

7.2.1.2 Set History To KEEP_ALL

The History QoS determines the behavior of a *DataWriter* or *DataReader* when its internal buffer fills up. There are two kinds:

- **KEEP_ALL:**The middleware will attempt to keep all the DDS samples until they are acknowledged (when the *DataWriter's* History is **KEEP_ALL**), or taken by the application (when the *DataReader's* History is **KEEP_ALL**).
- **KEEP_LAST:**The middleware will discard the oldest DDS samples to make room for new DDS samples. When the *DataWriter's* History is **KEEP_LAST**, DDS samples are discarded when a new call to **write()** is performed. When the *DataReader's* History is **KEEP_LAST**, DDS samples in the receive buffer are discarded when new DDS samples are received. This kind of history is associated with a depth that indicates how many historical DDS samples to retain.

```
<datawriter_qos>
   <history>
       <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    ...
</datawriter_qos>
...
<datareader_qos>
   <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
    ...
</datareader_qos>
```

The above section of the QoS profile tells RTI to use the policy **KEEP_ALL** for both *DataReader* and *DataWriter*.

7.2.1.3 Controlling Middleware Resources

With the ResourceLimits QosPolicy, you have full control over the amount of memory used by the middleware. In the example below, we specify that both the reader and writer will store up to 10 DDS samples (if you use a History **kind** of **KEEP_LAST**, the values specified here must be consistent with the value specified in the History's **depth**).

```
<datawriter_qos>
   <resource_limits>
       <max_samples>10</max_samples>
   </resource_limits>
   ...
</datawriter_qos>
...
<datareader_qos>
   <resource_limits>
       <max_samples>2</max_samples>
   </resource_limits>
   ...
</datareader_qos>
```

The above section tells RTI to allocate a buffer of 10 DDS samples for the *DataWriter* and 2 for the *DataReader*. If you do not specify any value for **max_samples**, the default behavior is for the middleware to allocate as much space as it needs.

One important function of the Resource Limits QoS policy, when used in conjunction with the Reliability and History QoS policies, is to govern how far "ahead" of its *DataReaders* a *DataWriter* may get before it will block, waiting for them to catch up. In many systems, consuming applications cannot acknowledge data as fast as its producing applications can put new data on the network. In such cases, the Resource Limits QoS policy provides a throttling mechanism that governs how many sent-but-not-yet-acknowledged DDS samples a *DataWriter* will maintain. If a *DataWriter* is configured for reliable **KEEP_ALL** operation, and it exceeds **max_samples** or **max_samples_per_instance**, calls to **write()** will block until the writer receives acknowledgements that will allow it to reclaim that memory.

The write operation can also block if the send window, configurable using the **max/min_send_window_size** fields in the DDS_RtpsReliableWriterProtocol_t structure in the DataWriter Protocol policy, fills up. The send window provides an alternative throttling mechanism that works with both KEEP_ALL and KEEP_LAST configurations.

If you see that your reliable publishing application is using an unacceptable amount of memory, you can specify a finite value for **max_samples**. By doing this, you restrain the size of the *DataWriter's* cache, causing it to use less memory; however, a smaller cache will fill more quickly, potentially causing the writer to block for a time when sending, decreasing throughput. If decreased throughput proves to be an issue, you can tune the reliability protocol to process acknowledgements and repairs more aggressively,

allowing the writer to clear its cache more effectively. A full discussion of the relevant reliability protocol parameters is beyond the scope of this example. However, you can find a useful example in **high_throughput.xml**. Also see the documentation for the DataReaderProtocol and DataWriterProtocol QoS policies in the on-line API documentation.

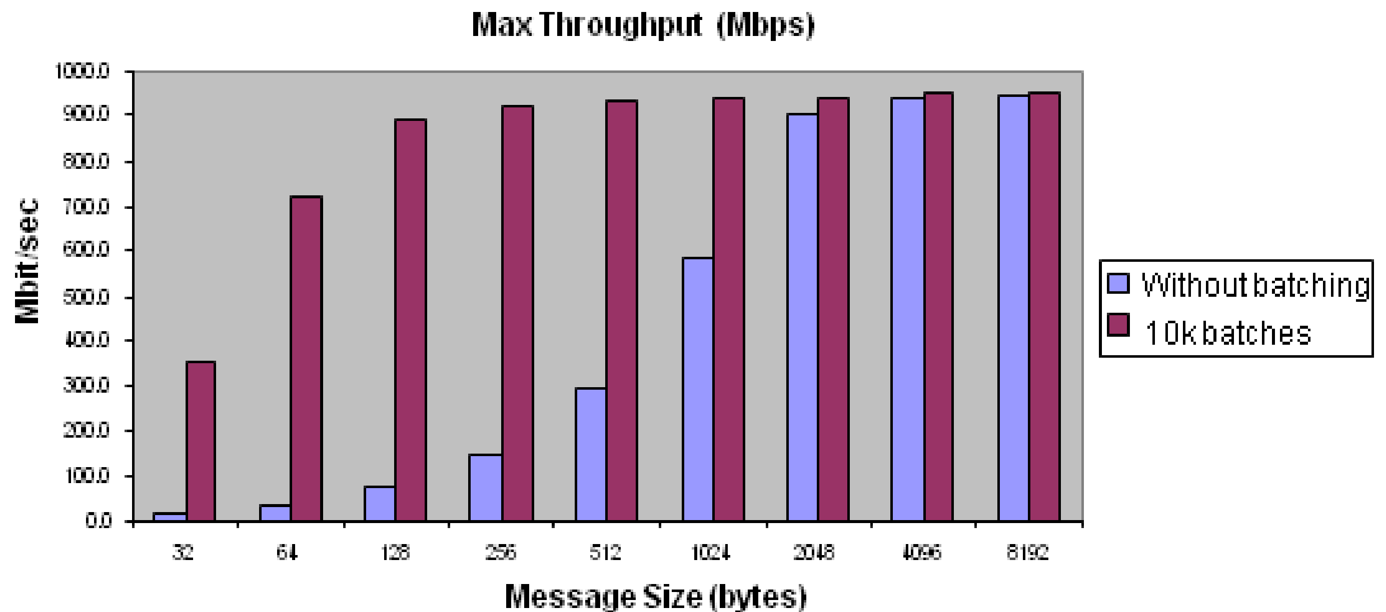
7.3 High Throughput for Streaming Data

This design pattern is useful for systems that produce a large number of small messages at a high rate.

In such cases, there is a small but measurable overhead in sending (and in the case of reliable communication, acknowledging) each message separately on the network. It is more efficient for the system to manage many DDS samples together as a group (referred to in the API as a batch) and then send the entire group in a single network packet. This allows *Connext DDS* to minimize the overhead of building a datagram and traversing the network stack.

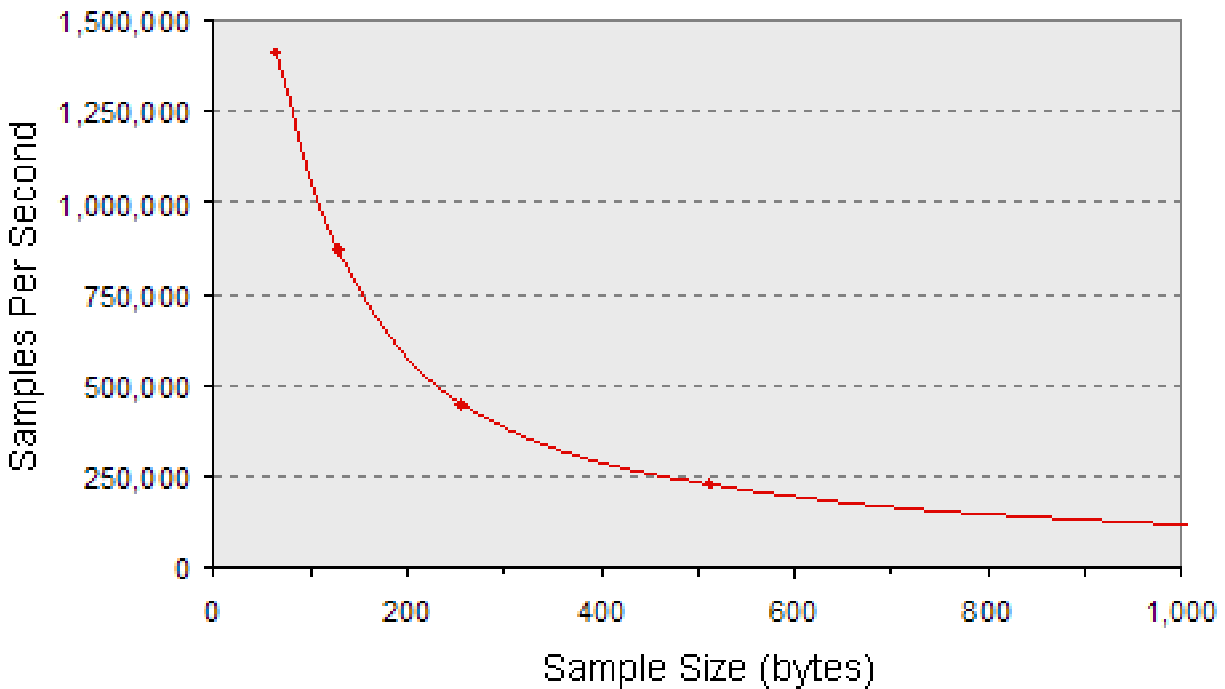
Batching increases throughput when writing small DDS samples at a high rate. As seen in Figure 7.3: Benefits of Batching: Sample Rates on the next page, throughput can be increased several-fold, much more closely approaching the physical limitations of the underlying network transport.

Figure 7.2: Benefits of Batching



Batching delivers tremendous benefits for messages of small size.

Figure 7.3: Benefits of Batching: Sample Rates



A subset of the batched throughput data above, expressed in terms of DDS samples per second.

Collecting DDS samples into a batch implies that they are not sent on the network (flushed) 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 share of the network's available bandwidth will be spent on acknowledgments and resending dropped data. In this case, reducing that meta-data overhead by turning on batching could decrease latency even while increasing throughput. Only an evaluation of your system's requirements and a measurement of its actual performance will indicate whether batching is appropriate. Fortunately, it is easy to enable and tune batching, as you will see below.

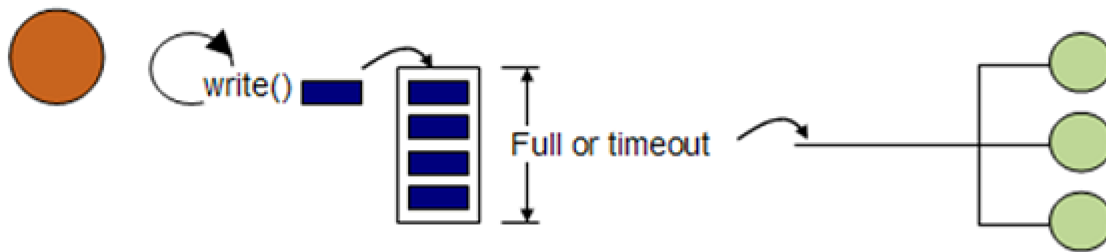
Batching is particularly useful when the system has to send a large number of small messages at a fast rate. Without this feature enabled, you may observe that your maximum throughput is less than the maximum bandwidth of your network. Simultaneously, you may observe high CPU loads. In this situation, the bottleneck in your system is the ability of the CPU to send packets through the OS network stack.

For example, in some algorithmic trading applications, market data updates arrive at a rate of from tens of thousands of messages per second to over a million; each update is some hundreds of bytes in size. It is often better to send these updates in batches than to publish them individually. Batching is also useful when sending a large number of small DDS samples over a connection where the bandwidth is severely constrained.

7.3.1 Implementation

RTI can automatically flush batches based on the maximum number of DDS samples, the total batch size, or elapsed time since the first DDS sample was placed in the batch, whichever comes first. Your application can also flush the current batch manually. Batching is completely transparent on the subscribing side; no special configuration is necessary.

Figure 7.4: Batching Implementation



RTI collects DDS samples in a batch until the batch is flushed.

For more information on batching, see the *RTI Connext DDS Core Libraries User's Manual* (Section 6.5.2) or API Reference HTML documentation (the Batch QosPolicy is described in the Infrastructure Module).

Using the batching feature is simple—just modify the QoS in the publishing application's configuration file.

For example, to enable batching with a batch size of 100 DDS samples, set the following QoS in your XML configuration file:

```
<datawriter_qos>
    ...
    <batch>
       <enable>true</enable>
       <max_samples>100</max_samples>
    </batch>
    ...
</datawriter_qos>
```

To enable batching with a maximum batch size of 8K bytes:

```
<datawriter_qos>
   ...
   <batch>
       <enable>true</enable>
       <max_data_bytes>8192</max_data_bytes>
   </batch>
   ...
</datawriter_qos>
```

To force the *DataWriter* to send whatever data is currently stored in a batch, use the *DataWriter*'s **flush()** operation.

7.4 Sending Large Data

If you have strict latency requirements, consider implementing *Zero Copy transfer over shared memory* or *RTI FlatData™ language binding*. Specifically, these features are recommended when your latency requirements cannot be met by regular C/C++ language binding (which defines the in-memory representation of a type), and the UDP and shared memory transports.

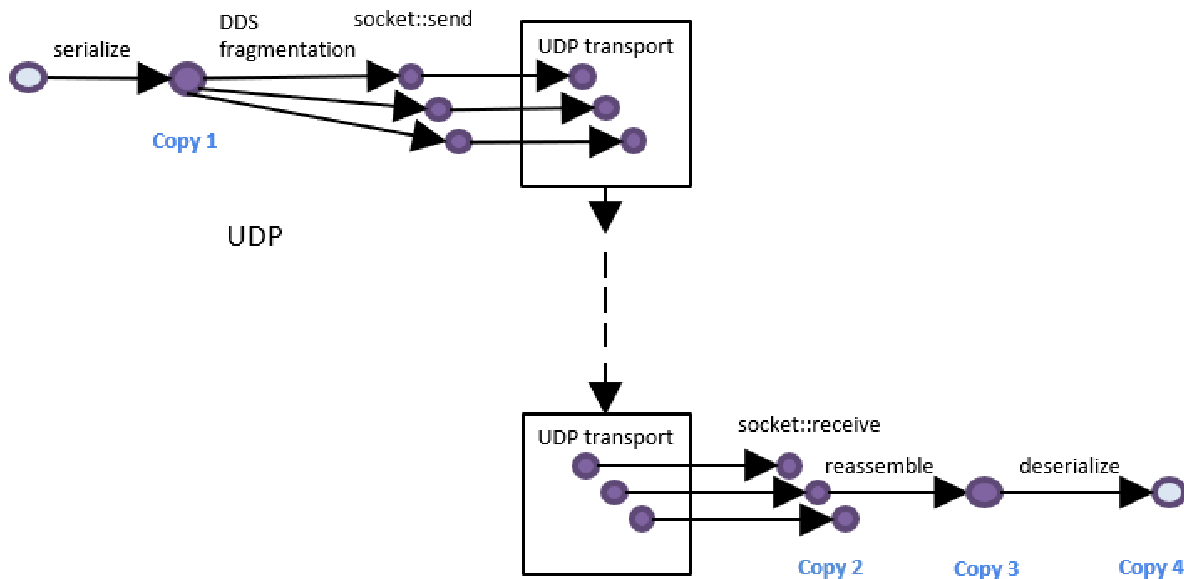
For example, video applications such as video conferencing, video surveillance, or computer vision usually have strict latency requirements, especially if the video signal is used to do control. Consider a latency requirement of less than 100 milliseconds. This latency must account for different components, such as video compression and decoding, transmission, image scaling, and application processing logic. Middleware, including *Connext DDS*, has its own latency budget.

Note: "Large data" means samples with a large serialized size, usually on the order of MBs, such as video frame samples. If you implement FlatData language binding or Zero Copy transfer over shared memory with data smaller than this, you may not see significant difference in latency or even pay a penalty in latency.

7.4.1 FlatData Language Binding

Figure 7.5: Number of Copies Out-of-the-Box on the next page shows how many times *Connext DDS* may copy a large sample sent over UDP or shared memory. The diagram assumes that the samples have to be fragmented because their serialized size is greater than the underlying transport MTU (maximum transmission unit). Note that these are copies in the middleware memory space—the operating system network stack may make additional copies.

Figure 7.5: Number of Copies Out-of-the-Box



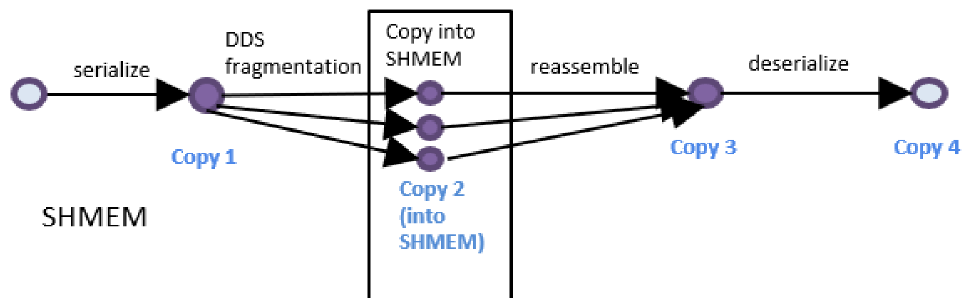
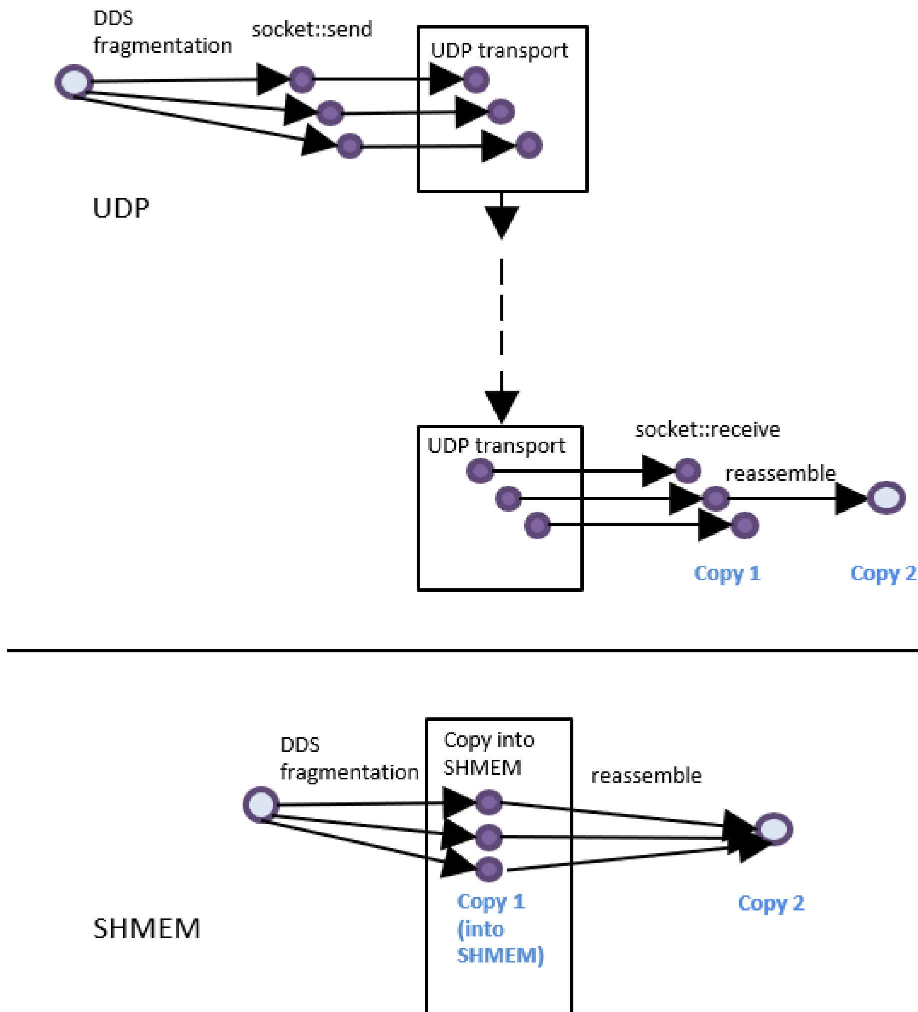


Figure 7.6: Number of Copies Using FlatData Language Binding on the next page shows that when using FlatData language binding, Copy 1 and Copy 4 in Figure 7.5: Number of Copies Out-of-the-Box above are removed for both UDP and shared memory (SHMEM) communications. FlatData is a language binding in which the in-memory representation of a sample matches the wire representation. Therefore, the cost of serialization/deserialization is zero. You can directly access the serialized data without deserializing it first.

Figure 7.6: Number of Copies Using FlatData Language Binding



For instructions on implementing FlatData language binding or Zero Copy transfer over shared memory, see the "Sending Large Data" chapter in the *RTI Connext DDS Core Libraries User's Manual*.

7.4.2 Zero Copy Transfer Over Shared Memory

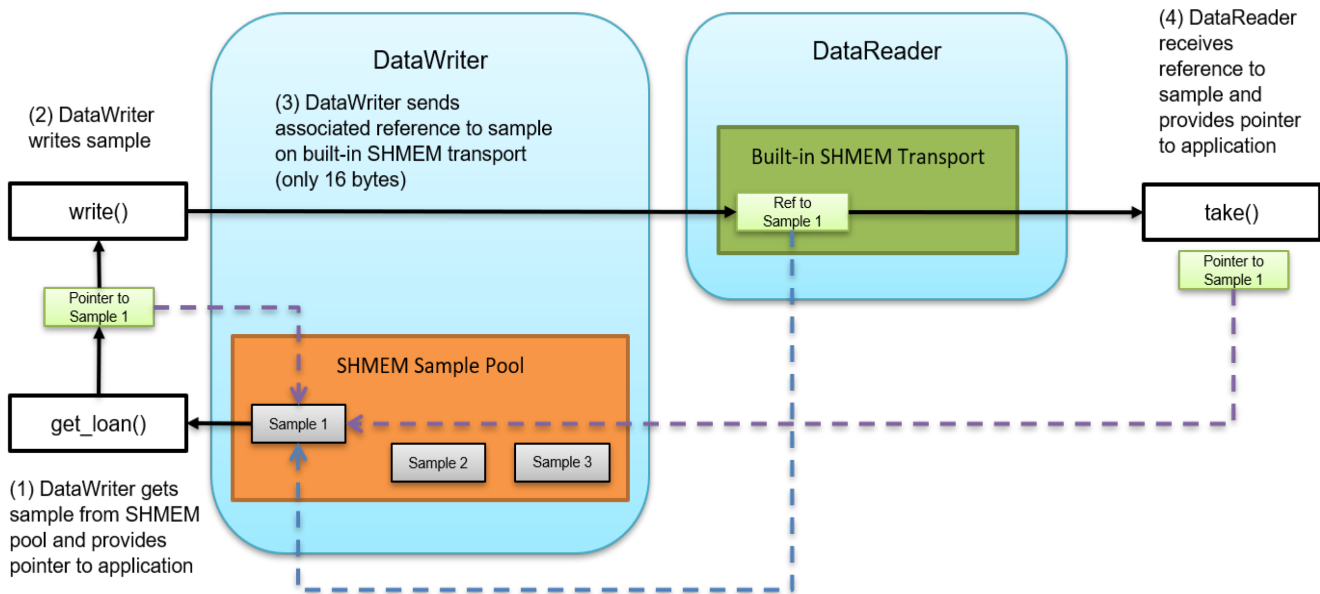
For communication within the same node using the built-in shared memory transport, by default *Connext DDS* copies a sample four times (see Figure 7.5: Number of Copies Out-of-the-Box on the previous page). FlatData language binding reduces the number of copies to two (see Figure 7.5: Number of Copies Out-of-the-Box on the previous page): the copy of the sample into the shared memory segment in the publishing application and the copy to reassemble the sample in the subscribing application. Two copies, however, may still be too many depending on the sample size and system requirements.

Zero Copy transfer over shared memory, provided as a separate library called *nddsmetp*, allows you to reduce the number of copies to zero for communications within the same host. This feature accomplishes zero copies by using the SHMEM built-in transport to send 16-byte references to samples within a

SHMEM segment owned by the *DataWriter*, instead of using the SHMEM built-in transport to send the serialized sample content by making a copy. See Figure 7.7: Zero Copy Transfer Over Shared Memory below.

With Zero Copy transfer over shared memory, there is no need for the *DataWriter* to serialize a sample, and there is no need for the *DataReader* to deserialize an incoming sample since the sample is accessed directly on the SHMEM segment created by the *DataWriter*.

Figure 7.7: Zero Copy Transfer Over Shared Memory



For instructions on implementing FlatData language binding or Zero Copy transfer over shared memory, see the "Sending Large Data" chapter in the *RTI Connext DDS Core Libraries User's Manual*.

7.4.3 Choosing between FlatData Language Binding and Zero Copy Transfer over Shared Memory

Whether to use Zero Copy transfer over shared memory or FlatData language binding, or both, depends on whether the *DataReaders* run on the same host as the *DataWriters*, on different hosts, or a combination of both. It also depends on the definition of the type. Zero Copy transfer over shared memory requires the FlatData language binding when the type is variable-size. The following table summarizes how to choose between these features:

Table 7.1 Zero Copy Transfer Over Shared Memory vs. FlatData Language Binding

	Readers and writers run on same host	Readers and writers run on different hosts	Some readers/writers run on same host, some on different hosts
Fixed-size types	Use Zero Copy	Use FlatData	Use both Zero Copy and FlatData

	Readers and writers run on same host	Readers and writers run on different hosts	Some readers/writers run on same host, some on different hosts
Variable-size types	Use both Zero Copy and FlatData	Use FlatData	Use both Zero Copy and FlatData

In summary, for *DataReaders* running on the same host as the *DataWriter*, the *DataWriter* can take advantage of Zero Copy transfer over shared memory. For *DataReaders* running on a different host, the *DataWriter* won't use Zero Copy transfer over shared memory, but can benefit from FlatData language binding. Therefore, when you have writers and readers running on the same and different hosts, it is recommended to use both Zero Copy transfer over shared memory and FlatData language binding, and let the *DataWriter* use the correct option for each *DataReader*.

For more information, see the "Sending Large Data" chapter in the *RTI Connext DDS Core Libraries User's Manual*.

7.5 Streaming Data over Unreliable Network Connections

Systems face unique challenges when sending data over lossy networks that also have high-latency and low-bandwidth constraints—for example, satellite and long-range radio links. While sending data over such a connection, a middleware tuned for a high-speed, dedicated Gigabit connection would throttle unexpectedly, cause unwanted timeouts and retransmissions, and ultimately suffer severe performance degradation.

For example, the transmission delay in satellite connections can be as much as 500 milliseconds to 1 second, which makes such a connection unsuitable for applications or middleware tuned for low-latency, real-time behavior. In addition, satellite links typically have lower bandwidth, with near-symmetric connection throughput of around 250–500 Kb/s and an advertised loss of approximately 3% of network packets. (Of course, the throughput numbers will vary based on the modem and the satellite service.) In light of these facts, a distributed application needs to tune the middleware differently when sending data over such networks.

Connext DDS is capable of maintaining liveliness and application-level QoS even in the presence of sporadic connectivity and packet loss at the transport level, an important benefit in mobile, or otherwise unreliable networks. It accomplishes this by implementing a reliable protocol that not only sequences and acknowledges application-level messages, but also monitors the liveliness of the link. Perhaps most importantly, it allows your application to fine-tune the behavior of this protocol to match the characteristics of your network. Without this latter capability, communication parameters optimized for more performant networks could cause communication to break down or experience unacceptable blocking times, a common problem in TCP-based solutions.

7.5.1 Implementation

When designing a system that demands reliability over a network that is lossy and has high latency and low throughput, it is critical to consider:

- How much data you send at one time (*e.g.,* your sample or batch size).
- How often you send it.
- The tuning of the reliability protocol for managing meta- and repair messages.

It is also important to be aware of whether your network supports multicast communication; if it does not, you may want to explicitly disable it in your middleware configuration (e.g., by using the **NDDS_ DISCOVERY_PEERS** environment variable or setting the initial_peers and **multicast_receive_address** in your Discovery QoS policy; see the API Reference HTML documentation).

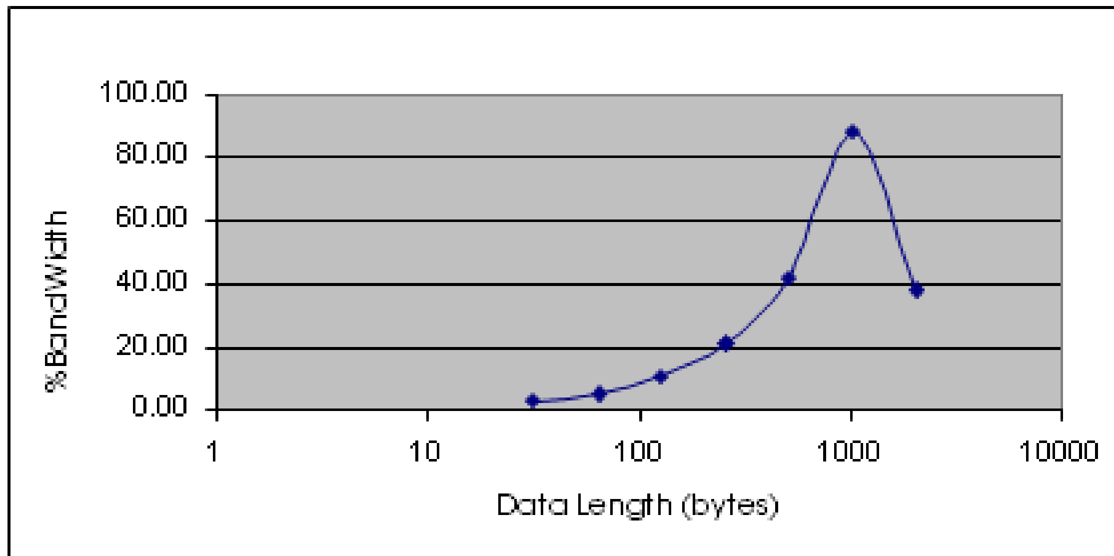
7.5.1.1 Managing Your Sample Size

Pay attention to your packet sizes to minimize or avoid IP-level fragmentation. Fragmentation can lead to additional repair meta-traffic that competes with the user traffic for bandwidth. Ethernet-like networks typically have a frame size of 1500 bytes; on such networks, sample sizes (or sample fragment sizes, if you've configured *Connext DDS* to fragment your DDS samples) should be kept to approximately 1400 bytes or less. Other network types will have different fragmentation thresholds.

The exact size of the DDS sample on the wire will depend not only on the size of your data fields, but also on the amount of padding introduced to ensure alignment while serializing the data.

Figure 7.8: Example Throughput Results over VSat Connection on the next page shows how an application's effective throughput (as a percentage of the theoretical capacity of the link) increases as the amount of data in each network packet increases. To put this relationship in another way: when transmitting a packet is expensive, it's advantageous to put as much data into it as possible. However, the trend reverses itself when the packet size becomes larger than the maximum transmission unit (MTU) of the physical network.

Figure 7.8: Example Throughput Results over VSat Connection



Correlation between sample size and bandwidth usage for a satellite connection with 3% packet loss ratio.

To understand why this occurs, remember that data is sent and received at the granularity of application DDS samples but dropped at the level of transport packets. For example, an IP datagram 10 KB in size must be fragmented into seven (1500-byte) Ethernet frames and then reassembled on the receiving end; the loss of any one of these frames will make reassembly impossible, leading to an effective loss, not of 1500 bytes, but of over 10 thousand bytes.

On an enterprise-class network, or even over the Internet, loss rates are very low, and therefore these losses are manageable. However, when loss rates reach several percent, the risk of losing at least one fragment in a large IP datagram becomes very large[a]. Over an unreliable protocol like UDP, such losses will eventually lead to near-total data loss as data size increases. Over a protocol like TCP, which provides reliability at the level of whole IP datagrams (not fragments), mounting losses will eventually lead to the network filling up with repairs, which will themselves be lost; the result can once again be near-total data loss.

To solve this problem, you need to repair data at the granularity at which it was lost: you need, not message-level reliability, but fragment-level reliability. This is an important feature of *Connext DDS*. When sending packets larger than the MTU of your underlying link, use RTI's data fragmentation and asynchronous publishing features to perform the fragmentation at the middleware level, hence relieving the IP layer of that responsibility.

---

[a]Suppose that a physical network delivers a 1 KB frame successfully 97% of the time. Now suppose that an application sends a 64 KB datagram. The likelihood that all fragments will arrive at their destination is 97% to the 64th power, or less than 15%.

```
<datawriter_qos>
    ...
    <publish_mode>
       <kind>ASYNCHRONOUS_PUBLISH_MODE_QOS</kind>
       <flow_controller_name>
               DDS_DEFAULT_FLOW_CONTROLLER_NAME
       </flow_controller_name>
    </publish_mode>
    ...
</datawriter_qos>
<participant_qos>
    ...
    <property>
       <value>
           <element>
               <name>
                   dds.transport.UDPv4.builtin.parent.message_size_max
               </name>
               <value>1500</value>
           </element>
       </value>
    </property>
    ...
</participant_qos>
```

The *DomainParticipant's* **dds.transport.UDPv4.builtin.parent.message_size_max** property sets the maximum size of a datagram that will be sent by the UDP/IPv4 transport. (If your application interfaces to your network over a transport other than UDP/IPv4, the name of this property will be different.) In this case, it is limiting all datagrams to the MTU of the link (assumed, for the sake of this example, to be equal to the MTU of Ethernet).

At the same time, the *DataWriter* is configured to send its DDS samples on the network, not synchronously when **write()** is called, but in a middleware thread. This thread will "flow" datagrams onto the network at a rate determined by the FlowController[a] identified by the flow_controller_name. In this case, the FlowController is a built-in instance that allows all data to be sent immediately. In a real-world application, you may want to use a custom FlowController that you create and configure in your application code. Further information on this topic is beyond the scope of this example. For more information on asynchronous publishing, see Section 6.4.1 in the *RTI Connext DDS Core Libraries User's Manual*. You can also find code examples demonstrating these capabilities online at the RTI Community website, accessible from https://community.rti.com. Navigate to **Examples** and search for Asynchronous Publication.

7.5.1.2 Acknowledge and Repair Efficiently

Piggyback heartbeat with each DDS sample. A *DataWriter* sends "heartbeats"—meta-data messages announcing available data and requesting acknowledgement—in two ways: periodically and "piggy-backed" into application data packets. Piggybacking heartbeats aggressively ensures that the middleware

[a]FlowControllers are not supported when using *Ada Language Support*.

will detect packet losses early, while allowing you to limit the number of extraneous network sends related to periodic heartbeats.

```
<datawriter_qos>
    ...
    <resource_limits>
        <!-- Used to configure piggybacks w/o batching -->
        <max_samples>
            20 <!-- An arbitrary finite size -->
        </max_samples>
    </resource_limits>

    <writer_resource_limits>
        <!-- Used to configure piggybacks w/ batching;
             see below -->
        <max_batches>
            20 <!-- An arbitrary finite size -->
        </max_batches>
    </writer_resource_limits>

    <protocol>
        <rtps_reliable_writer>
            <heartbeats_per_max_samples>
                20 <!-- Set same as max_samples -->
            </heartbeats_per_max_samples>
        </rtps_reliable_writer>
    </protocol>
    ...
</datawriter_qos>
```

The **heartbeats_per_max_samples** parameter controls how often the middleware will piggyback a heartbeat onto a data message: if the middleware is configured to cache 10 samples, for example, and **heartbeats_per_max_samples** is set to 5, a heartbeat will be piggybacked unto every other DDS sample. If **heartbeats_per_max_samples** is set equal to **max_samples**, this means that a heartbeat will be sent with each DDS sample.

7.5.1.3 Make Sure Repair Packets Don't Exceed Bandwidth Limitation

Applications can configure the maximum amount of data that a *DataWriter* will resend at a time using the **max_bytes_per_nack_response** parameter. For example, if a *DataReader* sends a negative acknowledgement (NACK) indicating that it missed 20 samples, each 10 KB in size, and **max_bytes_per_nack_response** is set to 100 KB, the *DataWriter* will only send the first 10 samples. The *DataReader* will have to NACK again to receive the remaining 10 samples.

In the following example, we limit the number of bytes so that we will never send more data than a 256 Kb/s, 1-ms latency link can handle over one second:

```
<datawriter_qos>
    ...
    <protocol>
```

```
        <rtps_reliable_writer>
            <max_bytes_per_nack_response>
                28000
            </max_bytes_per_nack_response>
        </rtps_reliable_writer>
    </protocol>
    ...
</datawriter_qos>
```

7.5.1.4 Use Batching to Maximize Throughput for Small Samples

If your application is sending data continuously, consider batching small samples to decrease the per-sample overhead. Be careful not to set your batch size larger than your link's MTU; see 7.5.1.1 Managing Your Sample Size on page 96.

For more information on how to configure throughput for small samples, see 7.3 High Throughput for Streaming Data on page 88.