OpenDDS Developer's Guide

OpenDDS Version 3.5

Supported by Object Computing, Inc. (OCI)

http://www.opendds.org

http://www.ociweb.com

Table of Contents

Preface

What Is OpenDDS?

OpenDDS is an open source implementation of two Object Management Group (OMG) specifications.

1) **Data Distribution Service (DDS) for Real-Time Systems v1.2** (OMG Document formal/07-01-01). This specification details the core functionality implemented by OpenDDS for real-time publish and subscribe applications and is described throughout this document.

2) **The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS) v2.1** (OMG Document formal/2010-11-01). This specification describes the primary requirements for interoperability between industry DDS implementations. This is not the only protocol for which a specifications exists, however, it is the protocol used for interoperability testing among DDS implementations.

OpenDDS is sponsored by Object Computing, Inc. (OCI) and is available via http://www.opendds.org/.

Licensing Terms

OpenDDS is made available under the *open source software* model. The source code may be freely downloaded and is open for inspection, review, comment, and improvement. Copies may be freely installed across all your systems and those of your customers. There is no charge for development or run-time licenses. The source code is designed to be compiled, and used, across a wide variety of hardware and operating systems architectures. You may modify it for your own needs, within the terms of the license agreements. You must not copyright OpenDDS software. For details of the licensing terms, see the file named `LICENSE` that is included in the OpenDDS source code distribution or visit http://www.opendds.org/license.html.

OpenDDS also utilizes other open source software products, including MPC (Make Project Creator), ACE (the ADAPTIVE Communication Environment), and TAO (The ACE ORB). More information about these products is available from OCI's web site at http://www.ociweb.com/products.

OpenDDS is open source and the development team welcomes contributions of code, tests, and ideas. Active participation by users ensures a robust implementation. Contact OCI if you are interested in contributing to the development of OpenDDS. Please note that any code that is contributed to and becomes part of the OpenDDS open source code base is subject to the same licensing terms as the rest of the OpenDDS code base.

About This Guide

This Developer's Guide corresponds to OpenDDS version 3.4. This guide is primarily focused on the specifics of using and configuring OpenDDS to build distributed, publish-subscribe applications. While it does give a general overview of the OMG Data Distribution Service, especially the Data-Centric Publish-Subscribe (DCPS) layer, this guide is not intended to provide comprehensive coverage of the specification. The intent of this guide is to help you become proficient with OpenDDS as quickly as possible.

Highlights of the OpenDDS 3.5 Release

- Updates to RTPS support resulting from both interoperability testing (March 2013 OMG meeting) and user feedback..
- Added config options to bind RTPS-related multicast sockets to specific network interfaces. See the ChangeLog for details.

- Generated TypeSupportImpl classes now contain nested typedefs that facilitate programming with C++ templates. See tests/DCPS/ManyTopicTest for an example of usage.
- Added a new option to opendds_idl, -Wb,v8, which generates type support for copying DCPS structs from C++ objects to JavaScript objects -- requires the V8 JavaScript engine. See https://npmjs.org/package/opendds for OpenDDS integration with Node.js.
- Fixed an opendds_idl code generated bug when typedefs of basic types are used as fields of structs.
- Fixed a bug in the DataReader relating to the Deadline timer.
- Fixed a bug in serialization with misaligned reads. It only impacts certain platforms with strict alignment requirements such as SPARC/SunCC.
- Fixed a bug in the rtps_udp transport, in certain cases an invalid Gap submessage was sent which can result in data samples not being received.
- Corrected a number of other bugs related to discovery and scaling.
- Clang 3.2 is now a supported compiler.

TAO Version Compatibility

OpenDDS 3.4 is compatible with the current patch levels of OCI TAO version 1.6a and 2.0a, as well as the current DOC Group beta/micro release. See the $DDS_ROOT/README file for details.

Conventions

This guide uses the following conventions:

`Fixed pitch text`	Indicates example code or information a user would enter using a keyboard.
`Bold fixed pitch text`	Indicates example code that has been modified from a previous example or text appearing in a menu or dialog box.
Italic text	Indicates a point of emphasis.
…	A horizontal ellipsis indicates that the statement is omitting text.
. . .	A vertical ellipsis indicates that a segment of code is omitted from the example.

Coding Examples

Throughout this guide, we illustrate topics with coding examples. The examples in this guide are intended for illustration purposes and should not be considered to be "production-ready" code. In particular, error handling is sometimes kept to a minimum to help the reader focus on the particular feature or technique that is being presented in the example. The source code for all these examples is available as part of the OpenDDS source code distribution in the `$DDS_ROOT/DevGuideExamples/` directory. MPC files are provided with the examples for generating build-tool specific files, such as GNU Makefiles or Visual C++ project and solution files. A Perl script named `run_test.pl` is provided with each example so you can easily run it.

Related Documents

Throughout this guide, we refer to various specifications published by the Object Management Group (OMG) and from other sources.

OMG references take the form *group/number* where *group* represents the OMG working group responsible for developing the specification, or the keyword `formal` if the specification has been formally adopted, and *number* represents the year, month, and serial number within the month the specification was released. For example, the OMG DDS version 1.2 specification is referenced as `formal/07-01-01`.

You can download any referenced OMG specification directly from the OMG web site by prepending `http://www.omg.org/cgi-bin/doc?` to the specification's reference. Thus, the specification `formal/07-01-01` becomes `http://www.omg.org/cgi-bin/doc?formal/07-01-01`. Providing this destination to a web browser should take you to a site from which you can download the referenced specification document.

Additional documentation on OpenDDS is produced and maintained by Object Computing, Inc. and is available from the *OpenDDS Community Portal* at `http://www.opendds.org`.

Here are some documents of interest and their locations:

Document	Location
Data Distribution Service (DDS) for Real-Time Systems v1.2 (OMG Document formal/07-01-01)	http://www.omg.org/cgi-bin/doc?formal/07-01-01
The Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS) v2.1 (OMG Document formal/2010-11-01)	http://www.omg.org/cgi-bin/doc?formal/10-11-01
OMG Data Distribution Portal	http://portals.omg.org/dds/
OpenDDS Buid Instructions, Architecture, and Doxygen Documentation	http://www.opendds.org/documentation.html
OpenDDS Frequently Asked Questions	http://www.opendds.org/faq.html

Supported Platforms

OCI regularly builds and tests OpenDDS on a wide variety of platforms, operating systems, and compilers. We continually update OpenDDS to support additional platforms. See the `$DDS_ROOT/README` file in the distribution for the most recent platform support information.

Customer Support

Enterprises are discovering that it takes considerable experience, knowledge, and money to design and build a complex distributed application that is robust and scalable. OCI can help you successfully architect and deliver your solution by drawing on the experience of seasoned architects who have extensive experience in today's middleware technologies and who understand how to leverage the power of DDS.

Our service areas include systems architecture, large-scale distributed application architecture, and object oriented design and development. We excel in technologies such as DDS (OpenDDS), CORBA (ACE+TAO, JacORB, and opalORB), Java EE (JBoss), FIX (QuickFIX), and FAST (QuickFAST).

Support offerings for OpenDDS include:

Consulting services to aid in the design of extensible, scalable, and robust publish-subscribe solutions, including the validation of domain-specific approaches, service selection, product customization and extension, and migrating your applications to OpenDDS from other publish-subscribe technologies and products.

24x7 support that guarantees the highest response level for your production-level systems.

On-demand service agreement for identification and assessment of minor bugs and issues that may arise during the development and deployment of OpenDDS-based solutions.

Our architects have specific and extensive domain expertise in security, telecommunications, defense, financial, and other real-time distributed applications.

We can provide professionals who can assist you on short-term engagements, such as architecture and design review, rapid prototyping, troubleshooting, and debugging. Alternatively, for larger engagements, we can provide mentors, architects, and programmers to work alongside your team, providing assistance and thought leadership throughout the life cycle of the project.

Contact us at +1.314.579.0066 or email <sales@ociweb.com> for more information.

Object Technology Training

OCI provides a rich program of more than 50 well-focused courses designed to give developers a solid foundation in a variety of technical topics, such as Object Oriented Analysis and Design, C++ Programming, Java Programming, Distributed Computing Technologies (including DDS), Patterns, XML, and UNIX/Linux. Our courses clearly explain major concepts and techniques, and demonstrate, through hands-on exercises, how they map to real-world applications.

Note *Our training offerings are constantly changing to meet the latest needs of our clients and to reflect changes in technology. Be sure to check out our web site at* http://www.ociweb.com *for updates to our Educational Programs.*

On-Site Classes

We can provide the following courses at your company's facility, integrating them seamlessly with other employee development programs. For more information about these or other courses in the OCI curriculum, visit our course catalog on-line at http://www.ociweb.com/training/.

Introduction to CORBA

In this one-day course, you will learn the benefits of distributed object computing; the role CORBA plays in developing distributed applications; when and where to apply CORBA; and future development trends in CORBA.

CORBA Programming with C++

In this hands-on, four-day course, you will learn: the role CORBA plays in developing distributed applications; the OMG's Object Management Architecture; how to write CORBA clients and servers in C++; how to use CORBA services such as Naming and Events; using CORBA exceptions; and basic and advanced features of the Portable Object Adapter (POA).

This course also covers the specification of interfaces using OMG Interface Definition Language (IDL) and details of the OMG IDL-to-C++ language mapping, and provides hands-on practice in developing CORBA clients and servers in C++ (using TAO).

Advanced CORBA Programming Using TAO

In this intensive, hands-on, four-day course, you will learn: several advanced CORBA concepts and techniques and how they are supported by TAO; how to configure TAO components for performance and space optimizations; and how to use TAO's various concurrency models to meet your application's end-to-end QoS guarantees. The course covers recent additions to the CORBA specifications and to TAO to support real-time CORBA programming, including Real-Time CORBA. It also covers TAO's Real-Time Event Service, Notification Service, and Implementation Repository, and provides extensive hands-on practice in developing advanced TAO clients and servers in C++. This course is intended for experienced and serious CORBA/C++ programmers.

Using the ACE C++ Framework

In this hands-on, four-day course, you will learn how to implement Interprocess Communication (IPC) mechanisms using the ACE (ADAPTIVE Communication Environment) IPC Service Access Point (SAP) classes and the Acceptor/Connector pattern. The course will also show you how to use a Reactor in event demultiplexing and dispatching; how to implement thread-safe applications using the ACE thread encapsulation class categories; and how to identify appropriate ACE components to use for your specific application needs.

Object-Oriented Design Patterns and Frameworks

In this three-day course, you will learn the critical language and terminology relating to design patterns, gain an understanding of key design patterns, learn how to select the appropriate pattern to apply in a given situation, and learn how to apply patterns to construct robust applications and frameworks. The course is designed for software developers who wish to utilize advanced object oriented design techniques and managers with a strong programming background who will be involved in the design and implementation of object oriented software systems.

OpenDDS Programming with C++

In this four-day course, you will learn to build applications using OpenDDS, the open source implementation of the OMG's Data Distribution Service (DDS) for Real-Time Systems. You will learn how to build data-centric systems that share data via OpenDDS. You will also learn to configure OpenDDS to meet your application's Quality of Service requirements. This course if intended for experienced C++ developers.

OpenDDS Modeling Software Development Kit (SDK)

In this two-day course, developers and architects gain hands-on experience using the OpenDDS Modeling SDK to design and build publish/subscribe applications that use OpenDDS. The Eclipse-based, open source Modeling SDK enables developers to define an application's middleware components and data structures as a UML model, then generate the code to implement the model using OpenDDS. The generated code can then be compiled and linked with the application to provide seamless middleware support to the application.

C++ Programming Using Boost

In this four-day course, you will learn about the most widely used and useful libraries that make up Boost. Students will learn how to easily apply these powerful libraries in their own development through detailed expert instructor-led training and by hands-on exercises. After finishing this course, class participants will be prepared to apply Boost to their project, enabling them to more quickly produce powerful, efficient, and platform independent applications.

Note *For information about training dates, contact us by phone at +1.314.579.0066, via electronic mail at* `training@ociweb.com`*, or visit our web site at* `http://www.ociweb.com` *to review the current course schedule.*

CHAPTER 1

Introduction

OpenDDS is an open source implementation of the OMG Data Distribution Service (DDS) for Real-Time Systems Specification v1.2 (OMG Document `formal/07-01-01`) and the Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS) v2.1 (OMG Document `formal/2010-11-01`). OpenDDS is sponsored by Object Computing, Inc. (OCI) and is available at http://www.opendds.org/. This developer's guide is based on the version 3.3 release of OpenDDS.

DDS defines a service for efficiently distributing application data between participants in a distributed application. This service is not specific to CORBA. The specification provides a Platform Independent Model (PIM) as well as a Platform Specific Model (PSM) that maps the PIM onto a CORBA IDL implementation. The service is divided into two levels of interfaces: the Data-Centric Publish-Subscribe (DCPS) layer and an optional Data Local Reconstruction Layer (DLRL). The DCPS layer transports data from publishers to subscribers according to Quality of Service constraints associated with the data topic, publisher, and subscriber. The DLRL allows distributed data to be shared by local objects located remotely from each other as if the data were local. The DLRL is built on top of the DCPS layer.

For additional details about DDS, developers should refer to the DDS specification (OMG Document `formal/07-01-01`) as it contains in-depth coverage of all the service's features. The scope of this document does not include a discussion of DLRL.

OpenDDS is the open-source C++ implementation of OMG's DDS specification developed and commercially supported by OCI. It is available for download from http://www.opendds.org/downloads.html and is compatible with the latest patch levels of OCI TAO version 1.6a and 2.0a, and the latest DOC release. OpenDDS version 2.4 was the last release that supported OCI TAO version 1.5a.

Note *OpenDDS currently implements the DCPS layer and is mostly compliant with the OMG DDS version 1.2 specification. None of the DLRL functionality is currently implemented. See the compliance information in or at http://www.opendds.org/ for more information.*

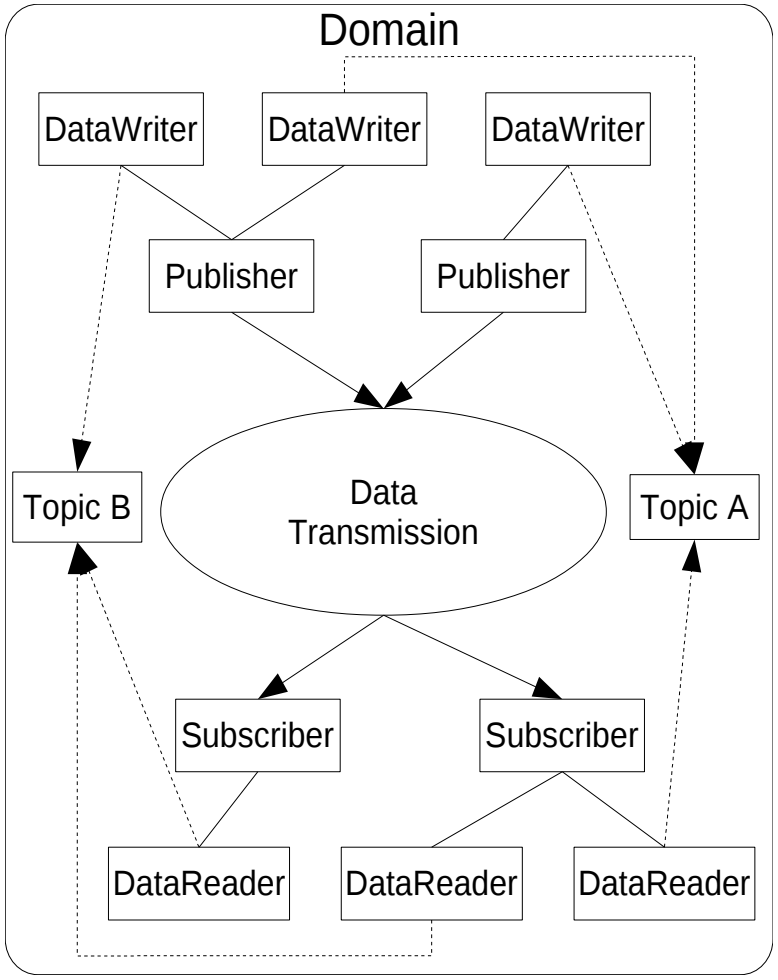
1.1. DCPS Overview

In this section we introduce the main concepts and entities of the DCPS layer and discuss how they interact and work together.

1.1.1 Basic Concepts

Figure 1-1 shows an overview of the DDS DCPS layer. The following subsections define the concepts shown in this diagram.

Figure 1-1 DCPS Conceptual Overview



1.1.1.1 Domain

The *domain* is the fundamental partitioning unit within DCPS. Each of the other entities belongs to a domain and can only interact with other entities in that same domain. Application code is free to interact with multiple domains but must do so via separate entities that belong to the different domains.

1.1.1.2 DomainParticipant

A *domain participant* is the entry-point for an application to interact within a particular domain. The domain participant is a factory for many of the objects involved in writing or reading data.

1.1.1.3 Topic

The *topic* is the fundamental means of interaction between publishing and subscribing applications. Each topic has a unique name within the domain and a specific data type that it publishes. Each topic data type can specify zero or more fields that make up its *key*. When publishing data, the publishing process always specifies the topic. Subscribers request data via the topic. In DCPS terminology you publish individual data *samples* for different *instances* on a topic. Each instance is associated with a unique value for the key. A publishing process publishes multiple data samples on the same instance by using the same key value for each sample.

1.1.1.4 DataWriter

The *data writer* is used by the publishing application code to pass values to the DDS. Each data writer is bound to a particular topic. The application uses the data writer's type-specific interface to publish samples on that topic. The data writer is responsible for marshaling the data and passing it to the publisher for transmission.

1.1.1.5 Publisher

The *publisher* is responsible for taking the published data and disseminating it to all relevant subscribers in the domain. The exact mechanism employed is left to the service implementation.

1.1.1.6 Subscriber

The *subscriber* receives the data from the publisher and passes it to any relevant data readers that are connected to it.

1.1.1.7 DataReader

The *data reader* takes data from the subscriber, demarshals it into the appropriate type for that topic, and delivers the sample to the application. Each data reader is bound to a particular topic. The application uses the data reader's type-specific interfaces to receive the samples.

1.1.2 Built-In Topics

The DDS specification defines a number of topics that are built-in to the DDS implementation. Subscribing to these *built-in topics* gives application developers access to the state of the domain being used including which topics are registered, which data readers and data writers are connected and disconnected, and the QoS settings of the various entities. While subscribed, the application receives samples indicating changes in the entities within the domain.

The following table shows the built-in topics defined within the DDS specification:

Table 1-1 Built-in Topics

Topic Name	Description
`DCPSParticipant`	Each instance represents a domain participant.
`DCPSTopic`	Each instance represents a normal (not built-in) topic.
`DCPSPublication`	Each instance represents a data writer.
`DCPSSubscription`	Each instance represents a data reader.

1.1.3 Quality Of Service Policies

The DDS specification defines a number of Quality of Service (QoS) policies that are used by applications to specify their QoS requirements to the service. Participants specify what behavior they require from the service and the service decides how to achieve these behaviors. These policies can be applied to the various DCPS entities (topic, data writer, data reader, publisher, subscriber, domain participant) although not all policies are valid for all types of entities.

Subscribers and publishers are matched using a request-versus-offered (RxO) model. Subscribers *request* a set of policies that are minimally required. Publishers *offer* a set of QoS policies to potential subscribers. The DDS implementation then attempts to match the requested policies with the offered policies; if these policies are compatible then the association is formed.

The QoS policies currently implemented by OpenDDS are discussed in detail in Chapter 3.

1.1.4 Listeners

The DCPS layer defines a callback interface for each entity that allows an application processes to “listen” for certain state changes or events pertaining to that entity. For example, a Data Reader Listener is notified when there are data values available for reading.

1.1.5 Conditions

Conditions and *Wait Sets* allow an alternative to listeners in detecting events of interest in DDS. The general pattern is

The application creates a specific kind of `Condition` object, such as a `StatusCondition`, and attaches it to a `WaitSet`.

- The application waits on the `WaitSet` until one or more conditions become true.
- The application calls operations on the corresponding entity objects to extract the necessary information.

- The `DataReader` interface also has operations that take a `ReadCondition` argument.
- `QueryCondition` objects are provided as part of the implementation of the Content-Subscription Profile. The `QueryCondition` interface extends the `ReadCondition` interface.

1.2 OpenDDS Implementation

1.2.1 Compliance

OpenDDS complies with the OMG DDS and the OMG DDS-RTPS specifications. Details of that compliance follows here.

1.2.1.1 DDS Compliance

Section 2 of the DDS specification defines five compliance points for a DDS implementation:

1) Minimum Profile

2) Content-Subscription Profile

3) Persistence Profile

4) Ownership Profile

5) Object Model Profile

As of version 2.2, OpenDDS complies with the entire DCPS layer of the DDS specification (including all optional profiles). This includes the implementation of all Quality of Service policies with the following notes:

- RELIABILITY.kind = RELIABLE is supported only if the TCP or IP Multicast transport (configured as reliable) is used or when the RTPS_UDP transport is used.
- TRANSPORT_PRIORITY is not implemented as changeable.

1.2.1.2 DDS-RTPS Compliance

The OpenDDS implementation complies with the requirements of the OMG DDS-RTPS specification.

OpenDDS RTPS Implementation Notes

The OMG DDS-RTPS specification (formal/2010-11-01) supplies statements for implementation, but not required for compliance. The following items should be taken into consideration when utilizing the OpenDDS RTPS functionality for transport selection and/or

discovery. Section numbers of the DDS-RTPS specification are supplied with each item for further reference.

Items not implemented in OpenDDS:

1) Non-default `LIVELINESS` QoS (8.7.2.2.3 and 8.4.13)

2) Sending fragmented data (8.4.14.1) (Receiving fragmented data is implemented)

3) Writer-side content filtering (8.7.3)
   OpenDDS may still drop samples that aren't needed (due to content filtering) by any associated readers -- this is done above the transport layer

4) Coherent sets for `PRESENTATION` QoS (8.7.5)

5) Directed writes (8.7.6)

6) Property lists (8.7.7)

7) Original writer info for `DURABLE` data (8.7.8) -- this would only be used for transient and persistent durability, which are not supported by the RTPS specification (8.7.2.2.1)

8) Key Hashes (8.7.9) are not generated, but they are optional

9) The `wait_for_acknowledgements()` method

10) `nackSuppressionDuration` (Table 8.47) and `heartbeatSuppressionDuration` (Table 8.62).

Note *Items 5 and 6 above are described in the DDS-RTPS specification. However, they do not have a corresponding concept in the DDS specification.*

1.2.2 OpenDDS Architecture

This section gives a brief overview of the OpenDDS implementation, its features, and some of its components. The `$DDS_ROOT` environment variable should point to the base directory of the OpenDDS distribution. Source code for OpenDDS can be found under the `$DDS_ROOT/dds/` directory. DDS tests can be found under `$DDS_ROOT/tests/`.

1.2.2.1 Design Philosophy

The OpenDDS implementation is based on a fairly strict interpretation of the OMG IDL PSM. In almost all cases the OMG's C++ Language Mapping for CORBA IDL is used to define how the IDL in the DDS specification is mapped into the C++ APIs that OpenDDS exposes to the client.

The main deviation from the OMG IDL PSM is that local interfaces are used for the entities and various other interfaces. These are defined as unconstrained (non-local) interfaces in the DDS specification. Defining them as local interfaces improves performance, reduces memory usage, simplifies the client's interaction with these interfaces, and makes it easier for clients to build their own implementations.

1.2.2.2 Extensible Transport Framework (ETF)

OpenDDS uses the IDL interfaces defined by the DDS specification to initialize and control service usage. Data transmission is accomplished via an OpenDDS-specific transport framework that allows the service to be used with a variety of transport protocols. This is referred to as *pluggable transports* and makes the extensibility of OpenDDS an important part of its architecture. OpenDDS currently supports TCP/IP, UDP/IP, IP multicast, shared-memory, and RTPS_UDP transport protocols as shown in Figure 1-2. Transports are typically specified via configuration files and are attached to various entities in the publisher and subscriber processes. Refer to Section 7.4.4 for details on configuring ETF components.

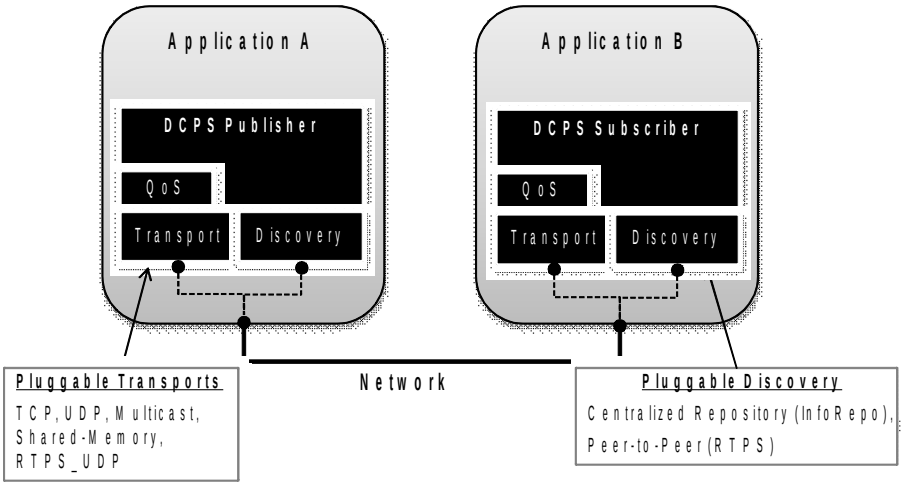


Figure 1-2 OpenDDS Extensible Transport Framework

The ETF enables application developers to implement their own customized transports. Implementing a custom transport involves specializing a number of classes defined in the transport framework. The `udp` transport provides a good foundation developers may use when creating their own implementation. See the `$DDS_ROOT/dds/DCPS/transport/udp/` directory for details.

1.2.2.3 DDS Discovery

DDS applications must discover one another via some central agent or through some distributed scheme. An important feature of OpenDDS is that DDS applications can be configured to perform discovery using the DCPSInfoRepo or RTPS discovery, but utilize a different transport type for data transfer between data writers and data readers. The OMG DDS specification (`formal/07-01-01`) leaves the details of discovery to the implementation. In the case of interoperability between DDS implementations, the OMG DDS-RTPS(`formal/2010-11-01`) specification provides requirements for a peer-to-peer style of discovery.

OpenDDS provides two options for discovery.

1) Information Repository: a centralized repository style that runs as a separate process allowing publishers and subscribers to discover one another centrally or

2) RTPS Discovery: a peer-to-peer style of discovery that utilizes the RTPS protocol to advertise availability and location information.

Interoperability with other DDS implementations must utilize the peer-to-peer method, but can be useful in OpenDDS-only deployments.

Centralized Discovery with DCPSInfoRepo

OpenDDS implements a standalone service called the DCPS Information Repository (DCPSInfoRepo) to achieve the centralized method. It is implemented as a CORBA server. When a client requests a subscription for a topic, the DCPS Information Repository locates the topic and notifies any existing publishers of the location of the new subscriber. The DCPSInfoRepo process needs to be running whenever OpenDDS is being used in a non-RTPS configuration. An RTPS configuration does not use the DCPSInfoRepo. The DCPSInfoRepo is not involved in data propagation, its role is limited in scope to OpenDDS applications discovering one another.

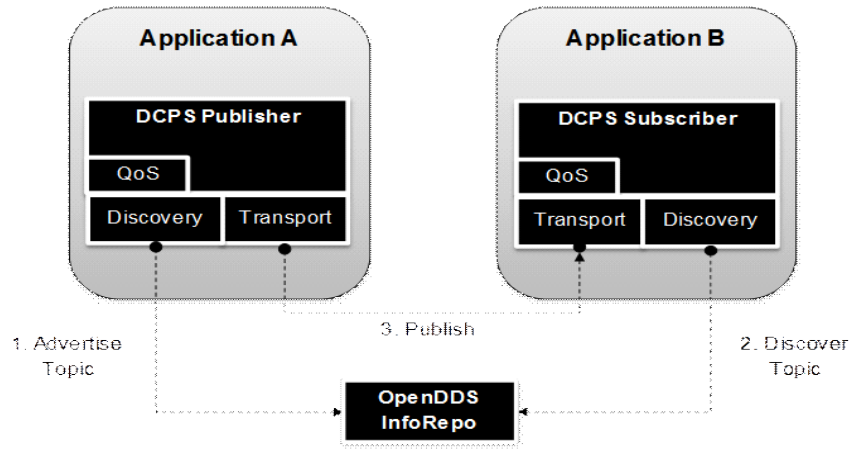


Figure 1-3 Centralized Discovery with OpenDDS InfoRepo

Application developers are free to run multiple information repositories with each managing their own non-overlapping sets of DCPS domains.

It is also possible to operate domains with more than a single repository, thus forming a distributed virtual repository. This is known as *Repository Federation*. In order for individual repositories to participate in a federation, each one must specify its own federation identifier value (a 32 bit numeric value) upon start-up. See 9.2 for further information about repository federations.

Peer-to-Peer Discovery with RTPS

DDS applications requiring a Peer-to-Peer discovery pattern can be accommodated by OpenDDS capabilities. This style of discovery is accomplished only through the use of the RTPS protocol as of the current release. This simple form of discovery is accomplished through simple configuration of DDS application data readers and data writers running in application processes as shown in Figure 1-3. As each participating process activates the DDS-RTPS discovery mechanisms in OpenDDS for their data readers and writers, network endpoints are created with either default or configured network ports such that DDS participants can begin advertising the availability of their data readers and data writers. After a period of time, those seeking one another based on criteria will find each other and establish a connection based on the configured pluggable transport as discussed in Extensible Transport Framework (ETF). A more detailed description of this flexible configuration approach is discussed in Section 7.4.1.1 and Section 7.4.5.5.

The following are additional implementation limits that developers need to take into consideration when developing and deploying applications that use RTPS discovery:

1) Domain IDs should be between 0 and 231 (inclusive) due to the way UDP ports are assigned to domain IDs. In each OpenDDS process, up to 120 domain participants are supported in each domain.

2) Topic names and type identifiers are limited to 256 characters.

3) OpenDDS's native multicast transport does not work with RTPS Discovery due to the way GUIDs are assigned (a warning will be issued if this is attempted).

For more details in how RTPS discovery occurs, a very good reference to read can be found in Section 8.5 of the Real-time Publish-Subscribe Wire Protocol DDS Interoperability Wire Protocol Specification (DDS-RTPS) v2.1 (OMG Document `formal/2010-11-01`).

1.2.2.4 Threading

OpenDDS creates its own ORB as well as a separate thread upon which to run that ORB. It also uses its own threads to process incoming and outgoing non-CORBA transport I/O. A separate thread is created to cleanup resources upon unexpected connection closure. Your application may get called back from these threads via the Listener mechanism of DCPS.

When publishing a sample via DDS, OpenDDS attempts to send the sample to any connected subscribers using the calling thread. If the send call blocks, then the sample may be queued for sending on a separate service thread. This behavior depends on the QoS policies described in Chapter 3.

All incoming data in the subscriber is read by a service thread and queued for reading by the application. DataReader listeners are called from the service thread.

1.2.2.5 Configuration

OpenDDS includes a file-based configuration framework for configuring both global items such as debug level, memory allocation, and discovery, as well as transport implementation details for publishers and subscribers. Configuration can also be achieved directly in code, however, it is recommended that configuration be externalized for ease of maintenance and reduction in runtime errors. The complete set of configuration options are described in Chapter 7.

1.3 Installation

The steps on how to build OpenDDS can be found in `$DDS_ROOT/INSTALL`.

To avoid compiling OpenDDS code that you will not be using, there are certain features than can be excluded from being built. The features are discussed below.

1.3.1 Building With A Feature Enabled Or Disabled

For the features described below, MPC is used for enabling (the default) a feature or disabling the feature. For a feature named *feature*, the following steps are used to disable the feature from the build:

1) Use the command line "features" argument to MPC:

```
mwc.pl -type <type> -features feature=0 DDS.mwc
```

Or alternatively, add the line `feature=0` to the file `$ACE_ROOT/bin/MakeProjectCreator/config/default.features` and regenerate the project files using MPC.

2) If you are using the `gnuace` MPC project type (which is the case if you will be using GNU make as your build system), add line "`feature=0`" to the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU`.

To explicitly enable the feature, use `feature=1` above.

Note *When creating a build from scratch, you can also use the* `$DDS_ROOT/configure` *script to enable or disable features. To disable the feature, pass* `--no-feature` *to the script, to enable pass* `--feature`*. In this case '-' is used instead of '_' in the feature name. For example, to disable feature* `content_subscription` *discussed below, pass* `--no-content-subscription` *to the configure script.*

1.3.2 Disabling The Building Of Built-In Topic Support

Feature Name: `built_in_topics`

Note *RTPS discovery must use Built-in Topic support, so this is not applicable in an RTPS discovery configuration.*

You can reduce the footprint of the core DDS library by up to 30% by disabling Built-in Topic Support. See Chapter 6 to determine if you build without BIT support.

1.3.3 Disabling The Building Of Compliance Profile Features

The DDS specification defines *compliance profiles* to provide a common terminology for indicating certain feature sets that a DDS implementation may or may not support. These

profiles are given below, along with the name of the MPC feature to use to disable support for that profile or components of that profile.

Many of profile options involve QoS settings. If you attempt to use a QoS value that is incompatible with a disabled profile, a runtime error will occur. If a profile involves a class, a compile time error will occur if you try to use the class and the profile is disabled.

1.3.3.1 Content-Subscription Profile

Feature Name: `content_subscription`

This profile adds the classes `ContentFilteredTopic`, `QueryCondition`, and `MultiTopic` discussed in Chapter 5.

In addition, individual classes can be excluded by using the features given in the table below.

Table 1-2: Content-Subscription Class Features

Class	Feature
ContentFilteredTopic	`content_filtered_topic`
QueryCondition	`query_condition`
MultiTopic	`multi_topic`

1.3.3.2 Persistence Profile

Feature Name: `persistence_profile`

This profile adds the QoS policy `DURABILITY_SERVICE` and the settings '`TRANSIENT`' and '`PERSISTENT`' of the `DURABILITY` QoS policy `kind`.

1.3.3.3 Ownership Profile

Feature Name: `ownership_profile`

This profile adds:

- the setting '`EXCLUSIVE`' of the `OWNERSHIP kind`
- support for the `OWNERSHIP_STRENGTH` policy
- setting a `depth > 1` for the `HISTORY` QoS policy.

Note *Currently, the OpenDDS code to support a* `HISTORY depth > 1` *is still enabled even if* `ownership_profile` *is disabled.*

1.3.3.4 Object Model Profile

Feature Name: `object_model_profile`

This profile includes support for the `PRESENTATION` access_scope setting of '`GROUP`'.

Note *Currently, the* `PRESENTATION` *access_scope of* '`TOPIC`' *is also excluded when* `object_model_profile` *is disabled.*

CHAPTER 2

Getting Started

2.1 Using DCPS

This chapter focuses on an example application using DCPS to distribute data from a single publisher process to a single subscriber process. It is based on a simple messenger application where a single publisher publishes messages and a single subscriber subscribes to them. We use the default QoS properties and the default TCP/IP transport. Full source code for this example may be found under the `$DDS_ROOT/DevGuideExamples/DCPS/Messenger/` directory. Additional DDS and DCPS features are discussed in later chapters.

2.1.1 Defining The Data Types

Each data type used by DDS is defined using IDL. OpenDDS uses `#pragma` directives to identify the data types that DDS transmits and processes. These data types are processed by the TAO IDL compiler and the OpenDDS IDL compiler to generate the necessary code to transmit data of these types with OpenDDS. Here is the IDL file that defines our Message data type:

```
module Messenger {

#pragma DCPS_DATA_TYPE "Messenger::Message"
```

```
#pragma DCPS_DATA_KEY "Messenger::Message subject_id"

  struct Message {
    string from;
    string subject;
    long subject_id;
    string text;
    long   count;
  };
};
```

The DCPS_DATA_TYPE pragma marks a data type for use with OpenDDS. A fully scoped type name must be used with this pragma. OpenDDS requires the data type to be a structure. The structure may contain scalar types (short, long, float, etc.), enumerations, strings, sequences, arrays, structures, and unions. This example defines the structure Message in the Messenger module for use in this OpenDDS example.

The DCPS_DATA_KEY pragma identifies a field of the DCPS data type that is used as the key for this type. A data type may have zero or more keys. These keys are used to identify different instances within a topic. Each key should be a numeric or enumerated type, a string, or a typedef of one of those types.[1] The pragma is passed the fully scoped type and the member name that identifies the key for that type. Multiple keys are specified with separate DCPS_DATA_KEY pragmas. In the above example, we identify the subject_id member of Messenger::Message as a key. Each sample published with a unique subject_id value will be defined as belonging to a different instance within the same topic. Since we are using the default QoS policies, subsequent samples with the same subject_id value are treated as replacement values for that instance.

2.1.2 Processing The IDL

The OpenDDS IDL is first processed by the TAO IDL compiler.

```
tao_idl Messenger.idl
```

In addition, we need to process the IDL file with the OpenDDS IDL compiler to generate the serialization and key support code that OpenDDS requires to marshal and demarshal the Message, as well as the type support code for the data readers and writers. This IDL compiler is located in $DDS_ROOT/bin/ and generates three files for each IDL file processed. The three files all begin with the original IDL file name and would appear as follows:

- *<filename>TypeSupport.idl*

1 Other types, such as structures, sequences, and arrays cannot be used directly as keys, though individual members of structs or elements of arrays can be used as keys when those members/elements are numeric, enumerated, or string types.

- *<filename>TypeSupportImpl.h*
- *<filename>TypeSupportImpl.cpp*

For example, running `opendds_idl` as follows

```
opendds_idl Messenger.idl
```

generates *MessengerTypeSupport.idl*, *MessengerTypeSupportImpl.h*, and *MessengerTypeSupportImpl.cpp*. The IDL file contains the `MessageTypeSupport`, `MessageDataWriter`, and `MessageDataReader` interface definitions. These are type-specific DDS interfaces that we use later to register our data type with the domain, publish samples of that data type, and receive published samples. The implementation files contain implementations for these interfaces. The generated IDL file should itself be compiled with the TAO IDL compiler to generate stubs and skeletons. These and the implementation file should be linked with your OpenDDS applications that use the Message type. The OpenDDS IDL compiler has a number of options that specialize the generated code. These options are described in Chapter 8.

Typically, you do not directly invoke the TAO or OpenDDS IDL compilers as above, but let your build environment do it for you. The entire process is simplified when using MPC, by inheriting from the dcpsexe_with_tcp project. Here is the MPC file section common to both the publisher and subscriber

```
project(*idl): dcps {
  // This project ensures the common components get built first.

  TypeSupport_Files {
    Messenger.idl
  }
  custom_only = 1
}
```

The dcps parent project adds the Type Support custom build rules. The TypeSupport_Files section above tells MPC to generate the Message type support files from `Messenger.idl` using the OpenDDS IDL complier. Here is the publisher section:

```
project(*Publisher) : dcpsexe_with_tcp {
  exename   = publisher
  after    += *idl

  TypeSupport_Files {
    Messenger.idl
  }

  Source_Files {
    Publisher.cpp
  }
}
```

The `dcpsexe_with_tcp` project links in the DCPS library.

For completeness, here is the subscriber section of the MPC file:

```
project(*Subscriber) : dcpsexe_with_tcp {

  exename   = subscriber
  after    += *idl

  TypeSupport_Files {
    Messenger.idl
  }

  Source_Files {
    Subscriber.cpp
    DataReaderListenerImpl.cpp
  }
}
```

2.1.3 A Simple Message Publisher

In this section we describe the steps involved in setting up a simple OpenDDS publication process. The code is broken into logical sections and explained as we present each section. We omit some uninteresting sections of the code (such as `#include` directives, error handling, and cross-process synchronization). The full source code for this sample publisher is found in the *`Publisher.cpp`* and *`Writer.cpp`* files in `$DDS_ROOT/DevGuideExamples/DCPS/Messenger/`.

2.1.3.1 Initializing the Participant

The first section of `main()` initializes the current process as an OpenDDS participant.

```
int main (int argc, char *argv[]) {
  try {
    DDS::DomainParticipantFactory_var dpf =
      TheParticipantFactoryWithArgs(argc, argv);
    DDS::DomainParticipant_var participant =
      dpf->create_participant(42, // domain ID
                              PARTICIPANT_QOS_DEFAULT,
                              0,  // No listener required
                              OpenDDS::DCPS::DEFAULT_STATUS_MASK);
    if (!participant) {
      std::cerr << "create_participant failed." << std::endl;
      return 1;
    }
```

The `TheParticipantFactoryWithArgs` macro is defined in `Service_Participant.h` and initializes the Domain Participant Factory with the command line arguments. These command line arguments are used to initialize the ORB that the OpenDDS service uses as well as the service itself. This allows us to pass `ORB_init()` options on the command line as well as OpenDDS configuration options of the form -DCPS*. Available OpenDDS options are fully described in Chapter 7.

The `create_participant()` operation uses the domain participant factory to register this process as a participant in the domain specified by the ID of 42. The participant uses

the default QoS policies and no listeners. Use of the OpenDDS default status mask ensures all relevant communication status changes (e.g., data available, liveliness lost) in the middleware are communicated to the application (e.g., via callbacks on listeners).

Users may define any number of domains using IDs in the range (0x0 ~ 0x7FFFFFFF). All other values are reserved for internal use by the implementation.

The Domain Participant object reference returned is then used to register our Message data type.

2.1.3.2 Registering the Data Type and Creating a Topic

First, we create a `MessageTypeSupportImpl` object, then register the type with a type name using the `register_type()` operation. In this example, we register the type with a nil string type name, which causes the `MessageTypeSupport` interface repository identifier to be used as the type name. A specific type name such as "*Message*" can be used as well.

```
Messenger::MessageTypeSupport_var mts =
  new Messenger::MessageTypeSupportImpl();
if (DDS::RETCODE_OK != mts->register_type(participant, "")) {
  std::cerr << "register_type failed." << std::endl;
  return 1;
}
```

Next, we obtain the registered type name from the type support object and create the topic by passing the type name to the participant in the `create_topic()` operation.

```
CORBA::String_var type_name = mts->get_type_name ();

DDS::Topic_var topic =
  participant->create_topic ("Movie Discussion List",
                             type_name,
                             TOPIC_QOS_DEFAULT,
                             0,   // No listener required
                             OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!topic) {
  std::cerr << "create_topic failed." << std::endl;
  return 1;
}
```

We have created a topic named "*Movie Discussion List*" with the registered type and the default QoS policies.

2.1.3.3 Creating a Publisher

Now, we are ready to create the publisher with the default publisher QoS.

```
DDS::Publisher_var pub =
  participant->create_publisher(PUBLISHER_QOS_DEFAULT,
                                0,    // No listener required
                                OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!pub) {
  std::cerr << "create_publisher failed." << std::endl;
```

```
  return 1;
}
```

2.1.3.4 Creating a DataWriter and Waiting for the Subscriber

With the publisher in place, we create the data writer.

```
// Create the datawriter
DDS::DataWriter_var writer =
  pub->create_datawriter(topic,
                         DATAWRITER_QOS_DEFAULT,
                         0,    // No listener required
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!writer) {
  std::cerr << "create_datawriter failed." << std::endl;
  return 1;
}
```

When we create the data writer we pass the topic object reference, the default QoS policies, and a null listener reference. We now narrow the data writer reference to a `MessageDataWriter` object reference so we can use the type-specific publication operations.

```
Messenger::MessageDataWriter_var message_writer =
     Messenger::MessageDataWriter::_narrow(writer);
```

The example code uses *conditions* and *wait* sets so the publisher waits for the subscriber to become connected and fully initialized. In a simple example like this, failure to wait for the subscriber may cause the publisher to publish its samples before the subscriber is connected.

The basic steps involved in waiting for the subscriber are:

1) Get the status condition from the data writer we created
2) Enable the Publication Matched status in the condition
3) Create a wait set
4) Attach the status condition to the wait set
5) Get the publication matched status
6) If the current count of matches is one or more, detach the condition from the wait set and proceed to publication
7) Wait on the wait set (can be bounded by a specified period of time)
8) Loop back around to step 5)

Here is the corresponding code:

```
// Block until Subscriber is available
DDS::StatusCondition_var condition =
    writer->get_statuscondition();
    condition>set_enabled_statuses(
     DDS::PUBLICATION_MATCHED_STATUS);

DDS::WaitSet_var ws = new DDS::WaitSet;
ws->attach_condition(condition);

while (true) {
  DDS::PublicationMatchedStatus matches;
  if (writer->get_publication_matched_status(matches)
      != DDS::RETCODE_OK) {
    std::cerr << "get_publication_matched_status failed!"
              << std::endl;
    return 1;
  }

  if (matches.current_count >= 1) {
    break;
  }

  DDS::ConditionSeq conditions;
  DDS::Duration_t timeout = { 60, 0 };
  if (ws->wait(conditions, timeout) != DDS::RETCODE_OK) {
    std::cerr << "wait failed!" << std::endl;
    return 1;
  }

}

ws->detach_condition(condition);
```

For more details about status, conditions, and wait sets, see Chapter 4.

2.1.3.5 Sample Publication

The message publication is quite straightforward:

```
// Write samples
Messenger::Message message;
message.subject_id = 99;
message.from       = "Comic Book Guy";
message.subject    = "Review";
message.text       = "Worst. Movie. Ever.";
message.count      = 0;
for (int i = 0; i < 10; ++i) {
  DDS::ReturnCode_t error = message_writer->write(message,
                                                  DDS::HANDLE_NIL);
  ++message.count;
  ++message.subject_id;
  if (error != DDS::RETCODE_OK) {
    // Log or otherwise handle the error condition
    return 1;
  }
}
```

For each loop iteration, calling `write()` causes a message to be distributed to all connected subscribers that are registered for our topic. Since the subject_id is the key for Message, each time subject_id is incremented and `write()` is called, a new instance is created (see 1.1.1.3). The second argument to `write()` specifies the instance on which we are publishing

the sample. It should be passed either a handle returned by `register_instance()` or `DDS::HANDLE_NIL`. Passing a `DDS::HANDLE_NIL` value indicates that the data writer should determine the instance by inspecting the key of the sample. See Section 2.2.1 for details on using instance handles during publication.

2.1.4 Setting Up The Subscriber

Much of the subscriber's code is identical or analogous to the publisher that we just finished exploring. We will progress quickly through the similar parts and refer you to the discussion above for details. The full source code for this sample subscriber is found in the *`Subscriber.cpp`* and *`DataReaderListener.cpp`* files in `$DDS_ROOT/DevGuideExamples/DCPS/Messenger/`.

2.1.4.1 Initializing the Participant

The beginning of the subscriber is identical to the publisher as we initialize the service and join our domain:

```
int main (int argc, char *argv[])
{
  try {
    DDS::DomainParticipantFactory_var dpf =
      TheParticipantFactoryWithArgs(argc, argv);
    DDS::DomainParticipant_var participant =
      dpf->create_participant(42, // Domain ID
                              PARTICIPANT_QOS_DEFAULT,
                              0,  // No listener required
                              OpenDDS::DCPS::DEFAULT_STATUS_MASK);
    if (!participant) {
      std::cerr << "create_participant failed." << std::endl;
      return 1 ;
    }
```

2.1.4.2 Registering the Data Type and Creating a Topic

Next, we initialize the message type and topic. Note that if the topic has already been initialized in this domain with the same data type and compatible QoS, the `create_topic()` invocation returns a reference corresponding to the existing topic. If the type or QoS specified in our `create_topic()` invocation do not match that of the existing topic then the invocation fails. There is also a `find_topic()` operation our subscriber could use to simply retrieve an existing topic.

```
    Messenger::MessageTypeSupport_var mts =
      new Messenger::MessageTypeSupportImpl();
    if (DDS::RETCODE_OK != mts->register_type(participant, "")) {
      std::cerr << "Failed to register the MessageTypeSupport." << std::endl;
      return 1;
    }

    CORBA::String_var type_name = mts->get_type_name ();

    DDS::Topic_var topic =
      participant->create_topic("Movie Discussion List",
```

```
                                type_name,
                                TOPIC_QOS_DEFAULT,
                                0,   // No listener required
                                OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!topic) {
  std::cerr << "Failed to create_topic." << std::endl;
  return 1;
}
```

2.1.4.3 Creating the subscriber

Next, we create the subscriber with the default QoS.

```
// Create the subscriber
DDS::Subscriber_var sub =
  participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT,
                                 0,   // No listener required
                                 OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!sub) {
  std::cerr << "Failed to create_subscriber." << std::endl;
  return 1;
}
```

2.1.4.4 Creating a DataReader and Listener

We need to associate a listener object with the data reader we create, so we can use it to detect when data is available. The code below constructs the listener object. The `DataReaderListenerImpl` class is shown in the next subsection.

```
DDS::DataReaderListener_var listener(new DataReaderListenerImpl);
```

The listener is allocated on the heap and assigned to a `DataReaderListener_var` object. This type provides reference counting behavior so the listener is automatically cleaned up when the last reference to it is removed. This usage is typical for heap allocations in OpenDDS application code and frees the application developer from having to actively manage the lifespan of the allocated objects.

Now we can create the data reader and associate it with our topic, the default QoS properties, and the listener object we just created.

```
// Create the Datareader
DDS::DataReader_var dr =
  sub->create_datareader(topic,
                         DATAREADER_QOS_DEFAULT,
                         listener,
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (!dr) {
  std::cerr << "create_datareader failed." << std::endl;
  return 1;
}
```

This thread is now free to perform other application work. Our listener object will be called on an OpenDDS thread when a sample is available.

2.1.5 The Data Reader Listener Implementation

Our listener class implements the `DDS::DataReaderListener` interface defined by the DDS specification. The `DataReaderListener` is wrapped within a `DCPS::LocalObject` which resolves ambiguously-inherited members such as `_narrow` and `_ptr_type`. The interface defines a number of operations we must implement, each of which is invoked to inform us of different events. The `OpenDDS::DCPS::DataReaderListener` defines operations for OpenDDS's special needs such as disconnecting and reconnected event updates. Here is the interface definition:

```
module DDS {
  local interface DataReaderListener : Listener {
    void on_requested_deadline_missed(in DataReader reader,
                                      in RequestedDeadlineMissedStatus status);
    void on_requested_incompatible_qos(in DataReader reader,
                                       in RequestedIncompatibleQosStatus status);
    void on_sample_rejected(in DataReader reader,
                            in SampleRejectedStatus status);
    void on_liveliness_changed(in DataReader reader,
                               in LivelinessChangedStatus status);
    void on_data_available(in DataReader reader);
    void on_subscription_matched(in DataReader reader,
                                 in SubscriptionMatchedStatus status);
    void on_sample_lost(in DataReader reader, in SampleLostStatus status);
  };
};
```

Our example listener class stubs out most of these listener operations with simple print statements. The only operation that is really needed for this example is `on_data_available()` and it is the only member function of this class we need to explore.

```
void DataReaderListenerImpl::on_data_available(DDS::DataReader_ptr reader)
{
  num_reads_ ++;

  try {
    Messenger::MessageDataReader_var reader_i =
          Messenger::MessageDataReader::_narrow(reader);
    if (!reader_i) {
      std::cerr << "read: _narrow failed." << std::endl;
      return;
    }
```

The code above narrows the generic data reader passed into the listener to the type-specific `MessageDataReader` interface. The following code takes the next sample from the message reader. If the take is successful and returns valid data, we print out each of the message's fields.

```
    Messenger::Message message;
    DDS::SampleInfo si ;
    DDS::ReturnCode_t status = reader_i->take_next_sample(message, si) ;

    if (status == DDS::RETCODE_OK) {

      if (si.valid_data == 1) {

          std::cout << "Message: subject    = " << message.subject.in() << std::endl
            << "         subject_id = " << message.subject_id   << std::endl
```

```
            << "         from       = " << message.from.in()    << std::endl
            << "         count      = " << message.count        << std::endl
            << "         text       = " << message.text.in()    << std::endl;
        }
        else if (si.instance_state == DDS::NOT_ALIVE_DISPOSED_INSTANCE_STATE)
        {
          std::cout << "instance is disposed" << std::endl;
        }
        else if (si.instance_state == DDS::NOT_ALIVE_NO_WRITERS_INSTANCE_STATE)
        {
          std::cout << "instance is unregistered" << std::endl;
        }
        else
        {
          std::cerr << "ERROR: received unknown instance state "
                    << si.instance_state << std::endl;
        }
      } else if (status == DDS::RETCODE_NO_DATA) {
          cerr << "ERROR: reader received DDS::RETCODE_NO_DATA!" << std::endl;
      } else {
          cerr << "ERROR: read Message: Error: " <<  status << std::endl;
      }
```

Note the sample read may contain invalid data. The valid_data flag indicates if the sample has valid data. There are two samples with invalid data delivered to the listener callback for notification purposes. One is the *dispose* notification, which is received when the `DataWriter` calls `dispose()` explicitly. The other is the *unregistered* notification, which is received when the `DataWriter` calls `unregister()` explicitly. The dispose notification is delivered with the instance state set to `NOT_ALIVE_DISPOSED_INSTANCE_STATE` and the unregister notification is delivered with the instance state set to `NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`.

If additional samples are available, the service calls this function again. However, reading values a single sample at a time is not the most efficient way to process incoming data. The Data Reader interface provides a number of different options for processing data in a more efficient manner. We discuss some of these operations in Section 2.2.

2.1.6 Cleaning Up In OpenDDS Clients

After we are finished in the publisher and subscriber, we can use the following code to clean up the OpenDDS-related objects:

```
participant->delete_contained_entities();
dpf->delete_participant(participant);
TheServiceParticipant->shutdown ();
```

The domain participant's `delete_contained_entities()` operation deletes all the topics, subscribers, and publishers created with that participant. Once this is done, we can use the domain participant factory to delete our domain participant.

Since the publication and subscription of data within DDS is decoupled, data is not guaranteed to be delivered if a publication is disassociated (shutdown) prior to all data that has been sent having been received by the subscriptions. If the application requires that all

published data be received, the `wait_for_acknowledgements()` operation is available to allow the publication to wait until all written data has been received. Data readers must have a `RELIABLE` setting for the `RELIABILITY` QoS (which is the default) in order for `wait_for_acknowledgements()` to work. This operation is called on individual `DataWriters` and includes a timeout value to bound the time to wait. The following code illustrates the use of `wait_for_acknowledgements()` to block for up to 15 seconds to wait for subscriptions to acknowledge receipt of all written data:

```
DDS::Duration_t shutdown_delay = {15, 0};
DDS::ReturnCode_t result;
result = writer->wait_for_acknowledgments(shutdown_delay);
if( result != DDS::RETCODE_OK) {
  std::cerr << "Failed while waiting for acknowledgment of "
            << "data being received by subscriptions, some data "
            << "may not have been delivered." << std::endl;
}
```

Note *The* `wait_for_acknowledgments()` *method is not implemented for applications using the RTPS transport. The operation will wait the entire time specified in the parameter before continuing.*

2.1.7 Running The Example

We are now ready to run our simple example. Running each of these commands in its own window should enable you to most easily understand the output.

First we will start a `DCPSInfoRepo` service so our publishers and subscribers can find one another.

Note *This step is not necessary if you are using peer-to-peer discovery by configuring your environment to use RTPS discovery.*

The `DCPSInfoRepo` executable is found in `$DDS_ROOT/bin/DCPSInfoRepo`. When we start the `DCPSInfoRepo` we need to ensure that publisher and subscriber application processes can also find the started `DCPSInfoRepo`. This information can be provided in one of three ways: a.) parameters on the command line , b.) generated and placed in a shared file for applications to use, or c.) parameters placed in a configuration file for other processes to use. For our simple example here we will use option ‘b’ by generating the location properties of the `DCPSInfoRepo` into a file so that our simple publisher and subscriber can read it in and connect to it.

From your current directory type:

Windows:

```
%DDS_ROOT%\bin\DCPSInfoRepo -o simple.ior
```

Unix:

```
$DDS_ROOT/bin/DCPSInfoRepo -o simple.ior
```

The `-o` parameter instructs the `DCPSInfoRepo` to generate its connection information to the file *`simple.ior`* for use by the publisher and subscriber. In a separate window navigate to the same directory that contains the *`simple.ior`* file and start the subscriber application in our example by typing:

Windows:

```
subscriber -DCPSInfoRepo file://simple.ior
```

Unix:

```
./subscriber -DCPSInfoRepo file://simple.ior
```

The command line parameters direct the application to use the specified file to locate the `DCPSInfoRepo`. Our subscriber is now waiting for messages to be sent, so we will now start the publisher in a separate window with the same parameters:

Windows:

```
publisher -DCPSInfoRepo file://simple.ior
```

Unix

```
./publisher -DCPSInfoRepo file://simple.ior
```

The publisher connects to the `DCPSInfoRepo` to find the location of any subscribers and begins to publish messages as well as write them to the console. In the subscriber window, you should also now be seeing console output from the subscriber that is reading messages from the topic demonstrating a simple publish and subscribe application.

You can read more about configuring your application for RTPS and other more advanced configuration options in Section 7.3.3 and Section 7.4.5.5 . To read more about configuring and using the `DCPSInfoRepo` go to Section 7.3 and Chapter 9. To find more about setting and using QoS features that modify the behavior of your application read Chapter 3.

2.1.8 Running Our Example With RTPS

The prior OpenDDS example has demonstrated how to build and execute an OpenDDS application using basic OpenDDS configurations and centralized discovery using the `DCPSInfoRepo` service. The following details what is needed to run the same example using RTPS for discovery and with an interoperable transport. This is important in scenarios when your OpenDDS application needs to interoperate with a non-OpenDDS implementation of the DDS specification or if you do not want to use centralized discovery in your deployment of OpenDDS.

The coding and building of the Messenger example above is not changed for using RTPS, so you will not need to modify or rebuild your publisher and subscriber services. This is a strength of the OpenDDS architecture in that to enable the RTPS capabilities, it is an exercise of configuration. Chapter 7 will cover more details concerning the configuration of all the available transports including RTPS, however, for this exercise we will enable RTPS for the Messenger example using a configuration file that the publisher and subscriber will share.

Navigate to the directory where your publisher and subscriber have been built. Create a new text file named *rtps.ini* and populate it with the following content:

```
[common]
DCPSGlobalTransportConfig=$file
DCPSDefaultDiscovery=DEFAULT_RTPS

[transport/the_rtps_transport]
transport_type=rtps_udp
```

More details of configuration files are specified in upcoming chapters, but the two lines of interest are called out for setting the discovery method and the data transport protocol to RTPS.

Now lets re-run our example with RTPS enabled by starting the subscriber process first and then the publisher to begin sending data. It is best to start them in separate windows to see the two working separately.

Start the subscriber with the `-DCPSConfigFile` command line parameter to point to the newly created configuration file...

Windows:

```
subscriber -DCPSConfigFile rtps.ini
```

Unix:

```
./subscriber -DCPSConfigFile rtps.ini
```

Now start the publisher with the same parameter...

Windows:

```
publisher -DCPSConfigFile rtps.ini
```

Unix:

```
./publisher -DCPSConfigFile rtps.ini
```

Since there is no centralized discovery in the RTPS specification, there are provisions to allow for wait times to allow discovery to occur. The specification sets the default to 30 seconds. When the two above processes are started there may be up to a 30 second delay depending on how far apart they are started from each other. This time can be adjusted in OpenDDS configuration files discussed later Section 7.3.3.

Another side effect of using RTPS for a transport is in the situation where a `wait_for_acknowledgements()` operation is used by a publisher when awaiting an acknowledgement from subscribers that they have successfully received the data. See this example from *`Publisher.cpp`* in our example:

```
// Wait for samples to be acknowledged
DDS::Duration_t timeout = { 30, 0 };
if (message_writer->wait_for_acknowledgments(timeout) != DDS::RETCODE_OK) {
  ACE_ERROR_RETURN((LM_ERROR,
                    ACE_TEXT("ERROR: %N:%l: main() -")
                    ACE_TEXT(" wait_for_acknowledgments failed!\n")),
                   -1);
}
```

Because this method is not implemented in OpenDDS a publisher process will wait the entire length of whatever value is specified as a parameter before it completes this section of the code. In the case of our example, the publisher makes this call after sending all of the data and then waits for acknowledgements for 30 seconds. When running the example you may experience this wait time at the end of the run before the publisher process exits.

Because the architecture of OpenDDS allows for pluggable discovery and pluggable transports the two configuration entries called out in the *`rtps.ini`* file above can be changed independently with one using RTPS and the other not using RTPS (e.g. centralized discovery using `DCPSInfoRepo`). Setting them both to RTPS in our example makes this application fully interoperable with other non-OpenDDS implementations.

2.2 Data Handling Optimizations

2.2.1 Registering And Using Instances In The Publisher

The previous example implicitly specifies the instance it is publishing via the sample's data fields. When `write()` is called, the data writer queries the sample's key fields to determine the instance. The publisher also has the option to explicitly register the instance by calling `register_instance()` on the data writer:

```
Messenger::Message message;
message.subject_id = 99;
DDS::InstanceHandle_t handle =
   message_writer->register_instance(message);
```

After we populate the Message structure we called the register_instance() function to register the instance. The instance is identified by the subject_id value of 99 (because we earlier specified that field as the key).

We can later use the returned instance handle when we publish a sample:

```
DDS::ReturnCode_t ret = data_writer->write(message, handle);
```

Publishing samples using the instance handle may be slightly more efficient than forcing the writer to query for the instance and is much more efficient when publishing the first sample on an instance. Without explicit registration, the first write causes resource allocation by OpenDDS for that instance.

Because resource limitations can cause instance registration to fail, many applications consider registration as part of setting up the publisher and always do it when initializing the data writer.

2.2.2 Reading Multiple Samples

The DDS specification provides a number of operations for reading and writing data samples. In the examples above we used the `take_next_sample()` operation, to read the next sample and "take" ownership of it from the reader. The Message Data Reader also has the following take operations.

- `take()`—Take a sequence of up to max_samples values from the reader
- `take_instance()`—Take a sequence of values for a specified instance
- `take_next_instance()`—Take a sequence of samples belonging to the same instance, without specifying the instance.

There are also "read" operations corresponding to each of these "take" operations that obtain the same values, but leave the samples in the reader and simply mark them as read in the `SampleInfo`.

Since these other operations read a sequence of values, they are more efficient when samples are arriving quickly. Here is a sample call to `take()` that reads up to 5 samples at a time.

```
MessageSeq messages(5);
DDS::SampleInfoSeq sampleInfos(5);
DDS::ReturnCode_t status =
    message_dr->take(messages,
                     sampleInfos,
                     5,
                     DDS::ANY_SAMPLE_STATE,
                     DDS::ANY_VIEW_STATE,
                     DDS::ANY_INSTANCE_STATE);
```

The three state parameters potentially specialize which samples are returned from the reader. See the DDS specification for details on their usage.

2.2.3 Zero-Copy Read

The read and take operations that return a sequence of samples provide the user with the option of obtaining a copy of the samples (single-copy read) or a reference to the samples (zero-copy read). The zero-copy read can have significant performance improvements over the single-copy read for large sample types. Testing has shown that samples of 8KB or less do not gain much by using zero-copy reads but there is little performance penalty for using zero-copy on small samples.

The application developer can specify the use of the zero-copy read optimization by calling `take()` or `read()` with a sample sequence constructed with a max_len of zero. The message sequence and sample info sequence constructors both take max_len as their first parameter and specify a default value of zero. The following example code is taken from `DevGuideExamples/DCPS/Messenger_ZeroCopy/`:

```
Messenger::MessageSeq messages;
DDS::SampleInfoSeq    info;

// get references to the samples  (zero-copy read of the samples)
DDS::ReturnCode_t status = dr->take (messages,
                                     info,
                                     DDS::LENGTH_UNLIMITED,
                                     DDS::ANY_SAMPLE_STATE,
                                     DDS::ANY_VIEW_STATE,
                                     DDS::ANY_INSTANCE_STATE);
```

After both zero-copy takes/reads and single-copy takes/reads, the sample and info sequences' length are set to the number of samples read. For the zero-copy reads, the `max_len` is set to a `value >= length`.

Since the application code has asked for a zero-copy loan of the data, it must return that loan when it is finished with the data:

```
dr->return_loan (messages, info);
```

Calling `return_loan()` results in the sequences' `max_len` being set to 0 and its owns member set to false, allowing the same sequences to be used for another zero-copy read.

If the first parameter of the data sample sequence constructor and info sequence constructor were changed to a value greater than zero, then the sample values returned would be copies. When values are copied, the application developer has the option of calling `return_loan()`, but is not required to do so.

If the `max_len` (the first) parameter of the sequence constructor is not specified, it defaults to 0; hence using zero-copy reads. Because of this default, a sequence will automatically call `return_loan()` on itself when it is destroyed. To conform with the DDS specification and be portable to other implementations of DDS, applications should not rely on this automatic `return_loan()` feature.

The second parameter to the sample and info sequences is the maximum slots available in the sequence. If the `read()` or `take()` operation's `max_samples` parameter is larger than this value, then the maximum samples returned by `read()` or `take()` will be limited by this parameter of the sequence constructor.

Although the application can change the length of a zero-copy sequence, by calling the `length(len)` operation, you are advised against doing so because this call results in copying the data and creating a single-copy sequence of samples.

CHAPTER 3

Quality of Service

3.1 Introduction

The previous examples use default QoS policies for the various entities. This chapter discusses the QoS policies which are implemented in OpenDDS and the details of their usage. See the DDS specification for further information about the policies discussed in this chapter.

3.2 QoS Policies

Each policy defines a structure to specify its data. Each entity supports a subset of the policies and defines a QoS structure that is composed of the supported policy structures. The set of allowable policies for a given entity is constrained by the policy structures nested in its QoS structure. For example, the Publisher's QoS structure is defined in the specification's IDL as follows:

```
module DDS {
  struct PublisherQos {
   PresentationQosPolicy presentation;
   PartitionQosPolicy partition;
   GroupDataQosPolicy group_data;
   EntityFactoryQosPolicy entity_factory;
 };
};
```

Setting policies is as simple as obtaining a structure with the default values already set, modifying the individual policy structures as necessary, and then applying the QoS structure to an entity (usually when it is created). We show examples of how to obtain the default QoS policies for various entity types in Section 3.2.1.

Applications can change the QoS of any entity by calling the set_qos() operation on the entity. If the QoS is changeable, existing associations are removed if they are no longer compatible and new associations are added if they become compatible. The `DCPSInfoRepo` re-evaluates the QoS compatibility and associations according to the QoS specification. If the compatibility checking fails, the call to set_qos() will return an error. The association re-evaluation may result in removal of existing associations or addition of new associations.

If the user attempts to change a QoS policy that is immutable (not changeable), then `set_qos()` returns `DDS::RETCODE_IMMUTABLE_POLICY`.

A subset of the QoS policies are changeable. Some changeable QoS policies, such as `USER_DATA`, `TOPIC_DATA`, `GROUP_DATA`, `LIFESPAN`, `OWNERSHIP_STRENGTH`, `TIME_BASED_FILTER`, `ENTITY_FACTORY`, `WRITER_DATA_LIFECYCLE`, and `READER_DATA_LIFECYCLE`, do not require compatibility and association re-evaluation. The `DEADLINE` and `LATENCY_BUDGET` QoS policies require compatibility re-evaluation, but not for association. The `PARTITION` QoS policy does not require compatibility re-evaluation, but does require association re-evaluation. The DDS specification lists `TRANSPORT_PRIORITY` as changeable, but the OpenDDS implementation does not support dynamically modifying this policy.

3.2.1 Default QoS Policy Values

Applications obtain the default QoS policies for an entity by instantiating a QoS structure of the appropriate type for the entity and passing it by reference to the appropriate `get_default_entity_qos()` operation on the appropriate factory entity. (For example, you would use a domain participant to obtain the default QoS for a publisher or subscriber.) The following examples illustrate how to obtain the default policies for publisher, subscriber, topic, domain participant, data writer, and data reader.

```
// Get default Publisher QoS from a DomainParticipant:
DDS::PublisherQos pub_qos;
DDS::ReturnCode_t ret;
ret = domain_participant->get_default_publisher_qos(pub_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default publisher QoS" << std::endl;
}

// Get default Subscriber QoS from a DomainParticipant:
DDS::SubscriberQos sub_qos;
ret = domain_participant->get_default_subscriber_qos(sub_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default subscriber QoS" << std::endl;
}
```

```
// Get default Topic QoS from a DomainParticipant:
DDS::TopicQos topic_qos;
ret = domain_participant->get_default_topic_qos(topic_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default topic QoS" << std::endl;
}

// Get default DomainParticipant QoS from a DomainParticipantFactory:
DDS::DomainParticipantQos dp_qos;
ret = domain_participant_factory->get_default_participant_qos(dp_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default participant QoS" << std::endl;
}

// Get default DataWriter QoS from a Publisher:
DDS::DataWriterQos dw_qos;
ret = pub->get_default_datawriter_qos(dw_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default data writer QoS" << std::endl;
}

// Get default DataReader QoS from a Subscriber:
DDS::DataReaderQos dr_qos;
ret = sub->get_default_datareader_qos(dr_qos);
if (DDS::RETCODE_OK != ret) {
  std::cerr << "Could not get default data reader QoS" << std::endl;
}
```

The following tables summarize the default QoS policies for each entity type in OpenDDS to which policies can be applied.

Table 3-1 Default DomainParticipant QoS Policies

Policy	Member	Default Value
USER_DATA	value	(not set)
ENTITY_FACTORY	autoenable_created_entities	true

Table 3-2 Default Topic QoS Policies

Policy	Member	Default Value
TOPIC_DATA	value	(not set)
DURABILITY	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DURABILITY_SERVICE	service_cleanup_delay.sec service_cleanup_delay.nanosec history_kind history_depth max_samples max_instances max_samples_per_instance	DURATION_ZERO_SEC DURATION_ZERO_NSEC KEEP_LAST_HISTORY_QOS 1 LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
DEADLINE	period.sec period.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
LIVELINESS	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC

Policy	Member	Default Value
RELIABILITY	kind max_blocking_time.sec max_blocking_time.nanosec	BEST_EFFORT_RELIABILITY_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
DESTINATION_ORDER	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
HISTORY	kind depth	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
TRANSPORT_PRIORITY	value	0
LIFESPAN	duration.sec duration.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
OWNERSHIP	kind	SHARED_OWNERSHIP_QOS

Table 3-3 Default Publisher QoS Policies

Policy	Member	Default Value
PRESENTATION	access_scope coherent_access ordered_access	INSTANCE_PRESENTATION_QOS 0 0
PARTITION	name	(empty sequence)
GROUP_DATA	value	(not set)
ENTITY_FACTORY	autoenable_created_entities	true

Table 3-4 Default Subscriber QoS Policies

Policy	Member	Default Value
PRESENTATION	access_scope coherent_access ordered_access	INSTANCE_PRESENTATION_QOS 0 0
PARTITION	name	(empty sequence)
GROUP_DATA	value	(not set)
ENTITY_FACTORY	autoenable_created_entities	true

Table 3-5 Default DataWriter QoS Policies

Policy	Member	Default Value
DURABILITY	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DURABILITY_SERVICE	service_cleanup_delay.sec service_cleanup_delay.nanosec history_kind history_depth max_samples max_instances max_samples_per_instance	DURATION_ZERO_SEC DURATION_ZERO_NSEC KEEP_LAST_HISTORY_QOS 1 LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
DEADLINE	period.sec period.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC

Policy	Member	Default Value
LIVELINESS	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
RELIABILITY	kind max_blocking_time.sec max_blocking_time.nanosec	RELIABLE_RELIABILITY_QOS[2] 0 100000000 (100 ms)
DESTINATION_ORDER	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
HISTORY	kind depth	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
TRANSPORT_PRIORITY	value	0
LIFESPAN	duration.sec duration.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
USER_DATA	value	(not set)
OWNERSHIP	kind	SHARED_OWNERSHIP_QOS
OWNERSHIP_STRENGTH	value	0
WRITER_DATA_LIFECYCLE	autodispose_unregistered_instances	1

Table 3-6 Default DataReader QoS Policies

Policy	Member	Default Value
DURABILITY	kind service_cleanup_delay.sec service_cleanup_delay.nanosec	VOLATILE_DURABILITY_QOS DURATION_ZERO_SEC DURATION_ZERO_NSEC
DEADLINE	period.sec period.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
LATENCY_BUDGET	duration.sec duration.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
LIVELINESS	kind lease_duration.sec lease_duration.nanosec	AUTOMATIC_LIVELINESS_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
RELIABILITY	kind max_blocking_time.sec max_blocking_time.nanosec	BEST_EFFORT_RELIABILITY_QOS DURATION_INFINITY_SEC DURATION_INFINITY_NSEC
DESTINATION_ORDER	kind	BY_RECEPTION_TIMESTAMP_ DESTINATIONORDER_QOS
HISTORY	kind depth	KEEP_LAST_HISTORY_QOS 1
RESOURCE_LIMITS	max_samples max_instances max_samples_per_instance	LENGTH_UNLIMITED LENGTH_UNLIMITED LENGTH_UNLIMITED
USER_DATA	value	(not set)
OWNERSHIP	kind	SHARED_OWNERSHIP_QOS
TIME_BASED_FILTER	minimum_separation.sec minimum_separation.nanosec	DURATION_ZERO_SEC DURATION_ZERO_NSEC
READER_DATA_LIFECYCLE	autopurge_nowriter_samples_delay.sec autopurge_nowriter_samples_delay.nanosec autopurge_disposed_samples_delay.sec autopurge_disposed_samples_delay.nanosec	DURATION_INFINITY_SEC DURATION_INFINITY_NSEC DURATION_INFINITY_SEC DURATION_INFINITY_NSEC

2 For OpenDDS versions, up to 2.0, the default reliability kind for data writers is best effort. For versions 2.0.1 and later, this is changed to reliable (to conform to the DDS specification).

3.2.2 LIVELINESS

The LIVELINESS policy applies to the topic, data reader, and data writer entities via the liveliness member of their respective QoS structures. Setting this policy on a topic means it is in effect for all data readers and data writers on that topic. Below is the IDL related to the liveliness QoS policy:

```
enum LivelinessQosPolicyKind {
  AUTOMATIC_LIVELINESS_QOS,
  MANUAL_BY_PARTICIPANT_LIVELINESS_QOS,
  MANUAL_BY_TOPIC_LIVELINESS_QOS
};

struct LivelinessQosPolicy {
  LivelinessQosPolicyKind kind;
  Duration_t lease_duration;
};
```

The LIVELINESS policy controls when and how the service determines whether participants are alive, meaning they are still reachable and active. The kind member setting indicates whether liveliness is asserted automatically by the service or manually by the specified entity. A setting of AUTOMATIC_LIVELINESS_QOS means that the service will send a liveliness indication if the participant has not sent any network traffic for the lease_duration. The MANUAL_BY_PARTICIPANT_LIVELINESS_QOS or MANUAL_BY_TOPIC_LIVELINESS_QOS setting means the specified entity (data writer for the “by topic” setting or domain participant for the “by participant” setting) must either write a sample or manually assert its liveliness within a specified heartbeat interval. The desired heartbeat interval is specified by the lease_duration member. The default lease duration is a pre-defined infinite value, which disables any liveliness testing.

To manually assert liveliness without publishing a sample, the application must call the assert_liveliness() operation on the data writer (for the “by topic” setting) or on the domain participant (for the “by participant” setting) within the specified heartbeat interval.

Data writers specify (*offer*) their own liveliness criteria and data readers specify (*request*) the desired liveliness of their writers. Writers that are not heard from within the lease duration (either by writing a sample or by asserting liveliness) cause a change in the LIVELINESS_CHANGED_STATUS communication status and notification to the application (e.g., by calling the data reader listener’s on_liveliness_changed() callback operation or by signaling any related wait sets).

This policy is considered during the establishment of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be established. Compatibility is determined by comparing the data reader’s requested liveliness with the data writer’s offered liveliness. Both the kind of liveliness

(automatic, manual by topic, manual by participant) and the value of the lease duration are considered in determining compatibility. The writer's offered kind of liveliness must be greater than or equal to the reader's requested kind of liveliness. The liveliness kind values are ordered as follows:

```
MANUAL_BY_TOPIC_LIVELINESS_QOS >
MANUAL_BY_PARTICIPANT_LIVELINESS_QOS >
AUTOMATIC_LIVELINESS_QOS
```

In addition, the writer's offered lease duration must be less than or equal to the reader's requested lease duration. Both of these conditions must be met for the offered and requested liveliness policy settings to be considered compatible and the association established.

3.2.3 RELIABILITY

The `RELIABILITY` policy applies to the topic, data reader, and data writer entities via the reliability member of their respective QoS structures. Below is the IDL related to the reliability QoS policy:

```
enum ReliabilityQosPolicyKind {
  BEST_EFFORT_RELIABILITY_QOS,
  RELIABLE_RELIABILITY_QOS
};

struct ReliabilityQosPolicy {
  ReliabilityQosPolicyKind kind;
  Duration_t max_blocking_time;
};
```

This policy controls how data readers and writers treat the data samples they process. The "best effort" value (`BEST_EFFORT_RELIABILITY_QOS`) makes no promises as to the reliability of the samples and could be expected to drop samples under some circumstances. The "reliable" value (`RELIABLE_RELIABILITY_QOS`) indicates that the service should eventually deliver all values to eligible data readers.

The `max_blocking_time` member of this policy is used when the history QoS policy is set to "keep all" and the writer is unable to return because of resource limits (due to transport backpressure—see 1.1.5 for details). When this situation occurs and the writer blocks for more than the specified time, then the write fails with a timeout return code. The default for this policy for data readers and topics is "best effort," while the default value for data writers is "reliable."

This policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an

association to be created. The reliability kind of data writer must be greater than or equal to the value of data reader.

3.2.4 HISTORY

The `HISTORY` policy determines how samples are held in the data writer and data reader for a particular instance. For data writers these values are held until the publisher retrieves them and successfully sends them to all connected subscribers. For data readers these values are held until "taken" by the application. This policy applies to the topic, data reader, and data writer entities via the history member of their respective QoS structures. Below is the IDL related to the history QoS policy:

```
enum HistoryQosPolicyKind {
  KEEP_LAST_HISTORY_QOS,
  KEEP_ALL_HISTORY_QOS
};

struct HistoryQosPolicy {
  HistoryQosPolicyKind kind;
  long depth;
};
```

The "keep all" value (`KEEP_ALL_HISTORY_QOS`) specifies that all possible samples for that instance should be kept. When "keep all" is specified and the number of unread samples is equal to the "resource limits" field of `max_samples_per_instance` then any incoming samples are rejected.

The "keep last" value (`KEEP_LAST_HISTORY_QOS`) specifies that only the last `depth` values should be kept. When a data writer contains depth samples of a given instance, a write of new samples for that instance are queued for delivery and the oldest unsent samples are discarded. When a data reader contains depth samples of a given instance, any incoming samples for that instance are kept and the oldest samples are discarded.

This policy defaults to a "keep last" with a `depth` of one.

3.2.5 DURABILITY

The `DURABILITY` policy controls whether data writers should maintain samples after they have been sent to known subscribers. This policy applies to the topic, data reader, and data writer entities via the durability member of their respective QoS structures. Below is the IDL related to the durability QoS policy:

```
enum DurabilityQosPolicyKind {
  VOLATILE_DURABILITY_QOS,         // Least Durability
  TRANSIENT_LOCAL_DURABILITY_QOS,
  TRANSIENT_DURABILITY_QOS,
  PERSISTENT_DURABILITY_QOS        // Greatest Durability
};
```

```
struct DurabilityQosPolicy {
  DurabilityQosPolicyKind kind;
};
```

By default the kind is VOLATILE_DURABILITY_QOS.

A durability kind of VOLATILE_DURABILITY_QOS means samples are discarded after being sent to all known subscribers. As a side effect, subscribers cannot recover samples sent before they connect.

A durability kind of TRANSIENT_LOCAL_DURABILITY_QOS means that data readers that are associated/connected with a data writer will be sent all of the samples in the data writer's history.

A durability kind of TRANSIENT_DURABILITY_QOS means that samples outlive a data writer and last as long as the process is alive. The samples are kept in memory, but are not persisted to permanent storage. A data reader subscribed to the same topic and partition within the same domain will be sent all of the cached samples that belong to the same topic/partition.

A durability kind of PERSISTENT_DURABILITY_QOS provides basically the same functionality as transient durability except the cached samples are persisted and will survive process destruction.

When transient or persistent durability is specified, the DURABILITY_SERVICE QoS policy specifies additional tuning parameters for the durability cache.

The durability policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be created. The durability kind value of the data writer must be greater than or equal to the corresponding value of the data reader. The durability kind values are ordered as follows:

```
PERSISTENT_DURABILITY_QOS >
TRANSIENT_DURABILITY_QOS >
TRANSIENT_LOCAL_DURABILITY_QOS >
VOLATILE_DURABILITY_QOS
```

3.2.6 DURABILITY_SERVICE

The DURABILITY_SERVICE policy controls deletion of samples in TRANSIENT or PERSISTENT durability cache. This policy applies to the topic and data writer entities via the durability_service member of their respective QoS structures and provides a way to specify HISTORY and RESOURCE_LIMITS for the sample cache. Below is the IDL related to the durability service QoS policy:

```
struct DurabilityServiceQosPolicy {
  Duration_t              service_cleanup_delay;
  HistoryQosPolicyKind    history_kind;
  long                    history_depth;
  long                    max_samples;
  long                    max_instances;
  long                    max_samples_per_instance;
};
```

The history and resource limits members are analogous to, although independent of, those found in the `HISTORY` and `RESOURCE_LIMITS` policies. The `service_cleanup_delay` can be set to a desired value. By default, it is set to zero, which means never clean up cached samples.

3.2.7 RESOURCE_LIMITS

The `RESOURCE_LIMITS` policy determines the amount of resources the service can consume in order to meet the requested QoS. This policy applies to the topic, data reader, and data writer entities via the resource_limits member of their respective QoS structures. Below is the IDL related to the resource limits QoS policy.

```
struct ResourceLimitsQosPolicy {
  long max_samples;
  long max_instances;
  long max_samples_per_instance;
};
```

The `max_samples` member specifies the maximum number of samples a single data writer or data reader can manage across all of its instances. The `max_instances` member specifies the maximum number of instances that a data writer or data reader can manage. The `max_samples_per_instance` member specifies the maximum number of samples that can be managed for an individual instance in a single data writer or data reader. The values of all these members default to unlimited (`DDS::LENGTH_UNLIMITED`).

Resources are used by the data writer to queue samples written to the data writer but not yet sent to all data readers because of backpressure from the transport. Resources are used by the data reader to queue samples that have been received, but not yet read/taken from the data reader.

3.2.8 PARTITION

The `PARTITION` QoS policy allows the creation of logical partitions within a domain. It only allows data readers and data writers to be associated if they have matched partition strings. This policy applies to the publisher and subscriber entities via the partition member of their respective QoS structures. Below is the IDL related to the partition QoS policy.

```
struct PartitionQosPolicy {
  StringSeq name;
```

```
};
```

The name member defaults to an empty sequence of strings. The default partition name is an empty string and causes the entity to participate in the default partition. The partition names may contain wildcard characters as defined by the POSIX `fnmatch` function (POSIX 1003.2-1992 section B.6).

The establishment of data reader and data writer associations depends on matching partition strings on the publication and subscription ends. Failure to match partitions is not considered a failure and does not trigger any callbacks or set any status values.

The value of this policy may be changed at any time. Changes to this policy may cause associations to be removed or added.

3.2.9 DEADLINE

The `DEADLINE` QoS policy allows the application to detect when data is not written or read within a specified amount of time. This policy applies to the topic, data writer, and data reader entities via the deadline member of their respective QoS structures. Below is the IDL related to the deadline QoS policy.

```
struct DeadlineQosPolicy {
  Duration_t period;
};
```

The default value of the `period` member is infinite, which requires no behavior. When this policy is set to a finite value, then the data writer monitors the changes to data made by the application and indicates failure to honor the policy by setting the corresponding status condition and triggering the `on_offered_deadline_missed()` listener callback. A data reader that detects that the data has not changed before the period has expired sets the corresponding status condition and triggers the `on_requested_deadline_missed()` listener callback.

This policy is considered during the creation of associations between data writers and data readers. The value of both sides of the association must be compatible in order for an association to be created. The deadline period of the data reader must be greater than or equal to the corresponding value of data writer.

The value of this policy may change after the associated entity is enabled. In the case where the policy of a data reader or data writer is made, the change is successfully applied only if the change remains consistent with the remote end of all associations in which the reader or writer is participating. If the policy of a topic is changed, it will affect only data readers and writers that are created after the change has been made. Any existing readers or writers, and any existing associations between them, will not be affected by the topic policy value change.

3.2.10 LIFESPAN

The LIFESPAN QoS policy allows the application to specify when a sample expires. Expired samples will not be delivered to subscribers. This policy applies to the topic and data writer entities via the lifespan member of their respective QoS structures. Below is the IDL related to the lifespan QoS policy.

```
struct LifespanQosPolicy {
  Duration_t duration;
}
```

The default value of the duration member is infinite, which means samples never expire. OpenDDS currently supports expired sample detection on the publisher side when using a DURABILITY kind other than VOLATILE. The current OpenDDS implementation may not remove samples from the data writer and data reader caches when they expire after being placed in the cache.

The value of this policy may be changed at any time. Changes to this policy affect only data written after the change.

3.2.11 USER_DATA

The USER_DATA policy applies to the domain participant, data reader, and data writer entities via the user_data member of their respective QoS structures. Below is the IDL related to the user data QoS policy:

```
struct UserDataQosPolicy {
  sequence<octet> value;
};
```

By default, the value member is not set. It can be set to any sequence of octets which can be used to attach information to the created entity. The value of the USER_DATA policy is available in respective built-in topic data. The remote application can obtain the information via the built-in topic and use it for its own purposes. For example, the application could attach security credentials via the USER_DATA policy that can be used by the remote application to authenticate the source.

3.2.12 TOPIC_DATA

The TOPIC_DATA policy applies to topic entities via the topic_data member of TopicQoS structures. Below is the IDL related to the topic data QoS policy:

```
struct TopicDataQosPolicy {
  sequence<octet> value;
};
```

By default, the `value` is not set. It can be set to attach additional information to the created topic. The value of the `TOPIC_DATA` policy is available in data writer, data reader, and topic built-in topic data. The remote application can obtain the information via the built-in topic and use it in an application-defined way.

3.2.13 GROUP_DATA

The `GROUP_DATA` policy applies to the publisher and subscriber entities via the group_data member of their respective QoS structures. Below is the IDL related to the group data QoS policy:

```
struct GroupDataQosPolicy {
  sequence<octet> value;
};
```

By default, the `value` member is not set. It can be set to attach additional information to the created entities. The value of the `GROUP_DATA` policy is propagated via built-in topics. The data writer built-in topic data contains the `GROUP_DATA` from the publisher and the data reader built-in topic data contains the `GROUP_DATA` from the subscriber. The `GROUP_DATA` policy could be used to implement matching mechanisms similar to those of the `PARTITION` policy described in 1.1.6 except the decision could be made based on an application-defined policy.

3.2.14 TRANSPORT_PRIORITY

The `TRANSPORT_PRIORITY` policy applies to topic and data writer entities via the transport_priority member of their respective QoS policy structures. Below is the IDL related to the TransportPriority QoS policy:

```
struct TransportPriorityQosPolicy {
  long value;
};
```

The default value member of `transport_priority` is zero. This policy is considered a hint to the transport layer to indicate at what priority to send messages. Higher values indicate higher priority. OpenDDS maps the priority value directly onto thread and DiffServ codepoint values. A default priority of zero will not modify either threads or codepoints in messages.

OpenDDS will attempt to set the thread priority of the sending transport as well as any associated receiving transport. Transport priority values are mapped from zero (default) through the maximum thread priority linearly without scaling. If the lowest thread priority is different from zero, then it is mapped to the transport priority value of zero. Where priority values on a system are inverted (higher numeric values are lower priority), OpenDDS maps these to an increasing priority value starting at zero. Priority values lower

than the minimum (lowest) thread priority on a system are mapped to that lowest priority. Priority values greater than the maximum (highest) thread priority on a system are mapped to that highest priority. On most systems, thread priorities can only be set when the process scheduler has been set to allow these operations. Setting the process scheduler is generally a privileged operation and will require system privileges to perform. On POSIX based systems, the system calls of `sched_get_priority_min()` and `sched_get_priority_max()` are used to determine the system range of thread priorities.

OpenDDS will attempt to set the DiffServ codepoint on the socket used to send data for the data writer if it is supported by the transport implementation. If the network hardware honors the codepoint values, higher codepoint values will result in better (faster) transport for higher priority samples. The default value of zero will be mapped to the (default) codepoint of zero. Priority values from 1 through 63 are then mapped to the corresponding codepoint values, and higher priority values are mapped to the highest codepoint value (63).

OpenDDS does not currently support modifications of the transport_priority policy values after creation of the data writer. This can be worked around by creating new data writers as different priority values are required.

3.2.15 LATENCY_BUDGET

The `LATENCY_BUDGET` policy applies to topic, data reader, and data writer entities via the latency_budget member of their respective QoS policy structures. Below is the IDL related to the LatencyBudget QoS policy:

```
struct LatencyBudgetQosPolicy {
  Duration_t duration;
};
```

The default value of `duration` is zero indicating that the delay should be minimized. This policy is considered a hint to the transport layer to indicate the urgency of samples being sent. OpenDDS uses the value to bound a delay interval for reporting unacceptable delay in transporting samples from publication to subscription. This policy is used for monitoring purposes only at this time. Use the `TRANSPORT_PRIORITY` policy to modify the sending of samples. The data writer policy value is used only for compatibility comparisons and if left at the default value of zero will result in all requested duration values from data readers being matched.

An additional listener extension has been added to allow reporting delays in excess of the policy duration setting. The `OpenDDS::DCPS::DataReaderListener` interface has an additional operation for notification that samples were received with a measured transport delay greater than the latency_budget policy duration. The IDL for this method is:

```
  struct BudgetExceededStatus {
```

```
  long total_count;
  long total_count_change;
  DDS::InstanceHandle_t last_instance_handle;
};

void on_budget_exceeded(
       in DDS::DataReader reader,
       in BudgetExceededStatus status);
```

To use the extended listener callback you will need to derive the listener implementation from the extended interface, as shown in the following code fragment:

```
class DataReaderListenerImpl
      : public virtual
        OpenDDS::DCPS::LocalObject<OpenDDS::DCPS::DataReaderListener>
```

Then you must provide a non-null implementation for the `on_budget_exceeded()` operation. Note that you will need to provide empty implementations for the following extended operations as well:

```
on_subscription_disconnected()
on_subscription_reconnected()
on_subscription_lost()
on_connection_deleted()
```

OpenDDS also makes the summary latency statistics available via an extended interface of the data reader. This extended interface is located in the `OpenDDS::DCPS` module and the IDL is defined as:

```
struct LatencyStatistics {
  GUID_t        publication;
  unsigned long n;
  double        maximum;
  double        minimum;
  double        mean;
  double        variance;
};

typedef sequence<LatencyStatistics> LatencyStatisticsSeq;

local interface DataReaderEx : DDS::DataReader {
  /// Obtain a sequence of statistics summaries.
  void get_latency_stats( inout LatencyStatisticsSeq stats);

  /// Clear any intermediate statistical values.
  void reset_latency_stats();

  /// Statistics gathering enable state.
  attribute boolean statistics_enabled;
};
```

To gather this statistical summary data you will need to use the extended interface. You can do so simply by dynamically casting the OpenDDS data reader pointer and calling the operations directly. In the following example, we assume that reader is initialized correctly by calling `DDS::Subscriber::create_datareader()`:

```
DDS::DataReader_var reader;
// ...

// To start collecting new data.
dynamic_cast<OpenDDS::DCPS::DataReaderImpl*>(reader.in())->
  reset_latency_stats();
dynamic_cast<OpenDDS::DCPS::DataReaderImpl*>(reader.in())->
  statistics_enabled(true);

// ...

// To collect data.
OpenDDS::DCPS::LatencyStatisticsSeq stats;
dynamic_cast<OpenDDS::DCPS::DataReaderImpl*>(reader.in())->
  get_latency_stats(stats);
for (unsigned long i = 0; i < stats.length(); ++i)
{
  std::cout << "stats[" << i << "]:" << std::endl;
  std::cout << "         n = " << stats[i].n << std::endl;
  std::cout << "       max = " << stats[i].maximum << std::endl;
  std::cout << "       min = " << stats[i].minimum << std::endl;
  std::cout << "      mean = " << stats[i].mean << std::endl;
  std::cout << "  variance = " << stats[i].variance << std::endl;
}
```

3.2.16 ENTITY_FACTORY

The `ENTITY_FACTORY` policy controls whether entities are automatically enabled when they are created. Below is the IDL related to the Entity Factory QoS policy:

```
struct EntityFactoryQosPolicy {
  boolean autoenable_created_entities;
};
```

This policy can be applied to entities that serve as factories for other entities and controls whether or not entities created by those factories are automatically enabled upon creation. This policy can be applied to the domain participant factory (as a factory for domain participants), domain participant (as a factory for publishers, subscribers, and topics), publisher (as a factory for data writers), or subscriber (as a factory for data readers). The default value for the `autoenable_created_entities` member is `true`, indicating that entities are automatically enabled when they are created. Applications that wish to explicitly enable entities some time after they are created should set the value of the `autoenable_created_entities` member of this policy to `false` and apply the policy to the appropriate factory entities. The application must then manually enable the entity by calling the entity's `enable()` operation.

The value of this policy may be changed at any time. Changes to this policy affect only entities created after the change.

3.2.17 PRESENTATION

The PRESENTATION QoS policy controls how changes to instances by publishers are presented to data readers. It affects the relative ordering of these changes and the scope of this ordering. Additionally, this policy introduces the concept of coherent change sets. Here is the IDL for the Presentation QoS:

```
enum PresentationQosPolicyAccessScopeKind {
  INSTANCE_PRESENTATION_QOS,
  TOPIC_PRESENTATION_QOS,
  GROUP_PRESENTATION_QOS
};

struct PresentationQosPolicy {
  PresentationQosPolicyAccessScopeKind access_scope;
  boolean coherent_access;
  boolean ordered_access;
};
```

The scope of these changes (access_scope) specifies the level in which an application may be made aware:

- INSTANCE_PRESENTATION_QOS (the default) indicates that changes occur to instances independently. Instance access essentially acts as a no-op with respect to coherent_access and ordered_access. Setting either of these values to true has no observable affect within the subscribing application.
- TOPIC_PRESENTATION_QOS indicates that accepted changes are limited to all instances within the same data reader or data writer.
- GROUP_PRESENTATION_QOS indicates that accepted changes are limited to all instances within the same publisher or subscriber.

Coherent changes (coherent_access) allow one or more changes to an instance be made available to an associated data reader as a single change. If a data reader does not receive the entire set of coherent changes made by a publisher, then none of the changes are made available. The semantics of coherent changes are similar in nature to those found in transactions provided by many relational databases. By default, coherent_access is false.

Changes may also be made available to associated data readers in the order sent by the publisher (ordered_access). This is similar in nature to the DESTINATION_ORDER QoS policy, however ordered_access permits data to be ordered independently of instance ordering. By default, ordered_access is false.

Note *This policy controls the ordering and scope of samples made available to the subscriber, but the subscriber application must use the proper logic in reading samples to guarantee the requested behavior. For more details, see Section 7.1.2.5.1.9 of the Version 1.2 DDS Specification.*

3.2.18 DESTINATION_ORDER

The DESTINATION_ORDER QoS policy controls the order in which samples within a given instance are made available to a data reader. If a history depth of one (the default) is specified, the instance will reflect the most recent value written by all data writers to that instance. Here is the IDL for the Destination Order Qos:

```
enum DestinationOrderQosPolicyKind {
  BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS,
  BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
};

struct DestinationOrderQosPolicy {
  DestinationOrderQosPolicyKind kind;
};
```

The BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS value (the default) indicates that samples within an instance are ordered in the order in which they were received by the data reader. Note that samples are not necessarily received in the order sent by the same data writer. To enforce this type of ordering, the BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS value should be used.

The BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS value indicates that samples within an instance are ordered based on a timestamp provided by the data writer. It should be noted that if multiple data writers write to the same instance, care should be taken to ensure that clocks are synchronized to prevent incorrect ordering on the data reader.

3.2.19 WRITER_DATA_LIFECYCLE

The WRITER_DATA_LIFECYCLE QoS policy controls the lifecycle of data instances managed by a data writer. Here is the IDL for the Writer Data Lifecycle QoS policy:

```
struct WriterDataLifecycleQosPolicy {
  boolean autodispose_unregistered_instances;
};
```

When autodispose_unregistered_instances is set to true (the default), a data writer disposes an instance when it is unregistered. In some cases, it may be desirable to prevent an instance from being disposed when an instance is unregistered. This policy could, for example, allow an EXCLUSIVE data writer to gracefully defer to the next data writer without affecting the instance state. Deleting a data writer implicitly unregisters all of its instances prior to deletion.

3.2.20 READER_DATA_LIFECYCLE

The READER_DATA_LIFECYCLE QoS policy controls the lifecycle of data instances managed by a data reader. Here is the IDL for the Reader Data Lifecycle QoS policy:

```
struct ReaderDataLifecycleQosPolicy {
  Duration_t autopurge_nowriter_samples_delay;
  Duration_t autopurge_disposed_samples_delay;
};
```

Normally, a data reader maintains data for all instances until there are no more associated data writers for the instance, the instance has been disposed, or the data has been taken by the user.

In some cases, it may be desirable to constrain the reclamation of these resources. This policy could, for example, permit a late-joining data writer to prolong the lifetime of an instance in fail-over situations.

The `autopurge_nowriter_samples_delay` controls how long the data reader waits before reclaiming resources once an instance transitions to the `NOT_ALIVE_NO_WRITERS` state. By default, `autopurge_nowriter_samples_delay` is infinite.

The `autopurge_disposed_samples_delay` controls how long the data reader waits before reclaiming resources once an instance transitions to the `NOT_ALIVE_DISPOSED` state. By default, `autopurge_disposed_samples_delay` is infinite.

3.2.21 TIME_BASED_FILTER

The `TIME_BASED_FILTER` QoS policy controls how often a data reader may be interested in changes in values to a data instance. Here is the IDL for the Time Based Filter QoS:

```
struct TimeBasedFilterQosPolicy {
  Duration_t minimum_separation;
};
```

An interval (`minimum_separation`) may be specified on the data reader. This interval defines a minimum delay between instance value changes; this permits the data reader to throttle changes without affecting the state of the associated data writer. By default, minimum_separation is zero, which indicates that no data is filtered. This QoS policy does not conserve bandwidth as instance value changes are still sent to the subscriber process. It only affects which samples are made available via the data reader.

3.2.22 OWNERSHIP

The `OWNERSHIP` policy controls whether more than one Data Writer is able to write samples for the same data-object instance. Ownership can be `EXCLUSIVE` or `SHARED`. Below is the IDL related to the Ownership QoS policy:

```
enum OwnershipQosPolicyKind {
  SHARED_OWNERSHIP_QOS,
  EXCLUSIVE_OWNERSHIP_QOS
};
```

```
struct OwnershipQosPolicy {
  OwnershipQosPolicyKind kind;
};
```

If the kind member is set to SHARED_OWNERSHIP_QOS, more than one Data Writer is allowed to update the same data-object instance. If the kind member is set to EXCLUSIVE_OWNERSHIP_QOS, only one Data Writer is allowed to update a given data-object instance (i.e., the Data Writer is considered to be the *owner* of the instance) and associated Data Readers will only see samples written by that Data Writer. The owner of the instance is determined by value of the OWNERSHIP_STRENGTH policy; the data writer with the highest value of strength is considered the owner of the data-object instance. Other factors may also influence ownership, such as whether the data writer with the highest strength is "alive" (as defined by the LIVELINESS policy) and has not violated its offered publication deadline constraints (as defined by the DEADLINE policy).

3.2.23 OWNERSHIP_STRENGTH

The OWNERSHIP_STRENGTH policy is used in conjunction with the OWNERSHIP policy, when the OWNERSHIP kind is set to EXCLUSIVE. Below is the IDL related to the Ownership Strength QoS policy:

```
struct OwnershipStrengthQosPolicy {
  long value;
};
```

The value member is used to determine which Data Writer is the *owner* of the data-object instance. The default value is zero.

3.3 Policy Example

The following sample code illustrates some policies being set and applied for a publisher.

```
DDS::DataWriterQos dw_qos;
pub->get_default_datawriter_qos (dw_qos);

dw_qos.history.kind = DDS::KEEP_ALL_HISTORY_QOS;

dw_qos.reliability.kind = DDS::RELIABLE_RELIABILITY_QOS;
dw_qos.reliability.max_blocking_time.sec = 10;
dw_qos.reliability.max_blocking_time.nanosec = 0;

dw_qos.resource_limits.max_samples_per_instance = 100;

DDS::DataWriter_var dw =
  pub->create_datawriter(topic,
                         dw_qos,
                         0,   // No listener
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

This code creates a publisher with the following qualities:

- `HISTORY` set to Keep All
- `RELIABILITY` set to Reliable with a maximum blocking time of 10 seconds
- The maximum samples per instance resource limit set to 100

This means that when 100 samples are waiting to be delivered, the writer can block up to 10 seconds before returning an error code. These same QoS settings on the Data Reader side would mean that up to 100 unread samples are queued by the framework before any are rejected. Rejected samples are dropped and the SampleRejectedStatus is updated.

CHAPTER 4

Conditions and Listeners

4.1 Introduction

The DDS specification defines two separate mechanisms for notifying applications of DCPS communication status changes. Most of the status types define a structure that contains information related to the change of status and can be detected by the application using conditions or listeners. The different status types are described in .

Each entity type (domain participant, topic, publisher, subscriber, data reader, and data writer) defines its own corresponding listener interface. Applications can implement this interface and then attach their listener implementation to the entity. Each listener interface contains an operation for each status that can be reported for that entity. The listener is asynchronously called back with the appropriate operation whenever a qualifying status change occurs. Details of the different listener types are discussed in 4.2 .

Conditions are used in conjunction with Wait Sets to let applications synchronously wait on events. The basic usage pattern for conditions involves creating the condition objects, attaching them to a wait set, and then waiting on the wait set until one of the conditions is triggered. The result of wait tells the application which conditions were triggered, allowing the application to take the appropriate actions to get the corresponding status information. Conditions are described in greater detail in 4.3 .

4.2 Communication Status Types

Each status type is associated with a particular entity type. This section is organized by the entity types, with the corresponding statuses described in subsections under the associated entity type.

Most of the statuses below are plain communication statuses. The exceptions are `DATA_ON_READERS` and `DATA_AVAILABLE` which are read statuses. Plain communication statuses define an IDL data structure. Their corresponding section below describes this structure and its fields. The read statuses are simple notifications to the application which then reads or takes the samples as desired.

Incremental values in the status data structure report a change since the last time the status was accessed. A status is considered accessed when a listener is called for that status or the status is read from its entity.

Fields in the status data structure with a type of `InstanceHandle_t` identify an entity (topic, data reader, data writer, etc.) by the instance handle used for that entity in the Built-In-Topics.

4.2.1 Topic Status Types

4.2.1.1 Inconsistent Topic Status

The `INCONSISTENT_TOPIC` status indicates that a topic was attempted to be registered that already exists with different characteristics. Typically, the existing topic may have a different type associated with it. The IDL associated with the Inconsistent Topic Status is listed below:

```
struct InconsistentTopicStatus {
  long total_count;
  long total_count_change;
};
```

The `total_count` value is the cumulative count of topics that have been reported as inconsistent. The `total_count_change` value is the incremental count of inconsistent topics since the last time this status was accessed.

4.2.2 Subscriber Status Types

4.2.2.1 Data On Readers Status

The `DATA_ON_READERS` status indicates that new data is available on some of the data readers associated with the subscriber. This status is considered a read status and does not

define an IDL structure. Applications receiving this status can call `get_datareaders()` on the subscriber to get the set of data readers with data available.

4.2.3 Data Reader Status Types

4.2.3.1 Sample Rejected Status

The `SAMPLE_REJECTED` status indicates that a sample received by the data reader has been rejected. The IDL associated with the Sample Rejected Status is listed below:

```
enum SampleRejectedStatusKind {
  NOT_REJECTED,
  REJECTED_BY_INSTANCES_LIMIT,
  REJECTED_BY_SAMPLES_LIMIT,
  REJECTED_BY_SAMPLES_PER_INSTANCE_LIMIT
};

struct SampleRejectedStatus {
  long total_count;
  long total_count_change;
  SampleRejectedStatusKind last_reason;
  InstanceHandle_t last_instance_handle;
};
```

The `total_count` value is the cumulative count of samples that have been reported as rejected. The `total_count_change` value is the incremental count of rejected samples since the last time this status was accessed. The `last_reason` value is the reason the most recently rejected sample was rejected. The `last_instance_handle` value indicates the instance of the last rejected sample.

4.2.3.2 Liveliness Changed Status

The `LIVELINESS_CHANGED` status indicates that there have been liveliness changes for one or more data writers that are publishing instances for this data reader. The IDL associated with the Liveliness Changed Status is listed below:

```
struct LivelinessChangedStatus {
  long alive_count;
  long not_alive_count;
  long alive_count_change;
  long not_alive_count_change;
  InstanceHandle_t last_publication_handle;
};
```

The `alive_count` value is the total number of data writers currently active on the topic this data reader is reading. The `not_alive_count` value is the total number of data writers writing to the data reader's topic that are no longer asserting their liveliness. The `alive_count_change` value is the change in the alive count since the last time the status was accessed. The `not_alive_count_change` value is the change in the not alive count since

the last time the status was accessed. The `last_publication_handle` is the handle of the last data writer whose liveliness has changed.

4.2.3.3 Requested Deadline Missed Status

The `REQUESTED_DEADLINE_MISSED` status indicates that the deadline requested via the Deadline QoS policy was not respected for a specific instance. The IDL associated with the Requested Deadline Missed Status is listed below:

```
struct RequestedDeadlineMissedStatus {
  long total_count;
  long total_count_change;
  InstanceHandle_t last_instance_handle;
};
```

The `total_count` value is the cumulative count of missed requested deadlines that have been reported. The `total_count_change` value is the incremental count of missed requested deadlines since the last time this status was accessed. The `last_instance_handle` value indicates the instance of the last missed deadline.

4.2.3.4 Requested Incompatible QoS Status

The `REQUESTED_INCOMPATIBLE_QOS` status indicates that one or more QoS policy values that were requested were incompatible with what was offered. The IDL associated with the Requested Incompatible QoS Status is listed below:

```
struct QosPolicyCount {
  QosPolicyId_t policy_id;
  long count;
};

typedef sequence<QosPolicyCount> QosPolicyCountSeq;

struct RequestedIncompatibleQosStatus {
  long total_count;
  long total_count_change;
  QosPolicyId_t last_policy_id;
  QosPolicyCountSeq policies;
};
```

The `total_count` value is the cumulative count of times data writers with incompatible QoS have been reported. The `total_count_change` value is the incremental count of incompatible data writers since the last time this status was accessed. The `last_policy_id` value identifies one of the QoS policies that was incompatible in the last incompatibility detected. The policies value is a sequence of values that indicates the total number of incompatibilities that have been detected for each QoS policy.

4.2.3.5 Data Available Status

The DATA_AVAILABLE status indicates that samples are available on the data writer. This status is considered a read status and does not define an IDL structure. Applications receiving this status can use the various take and read operations on the data reader to retrieve the data.

4.2.3.6 Sample Lost Status

The SAMPLE_LOST status indicates that a sample has been lost and never received by the data reader. The IDL associated with the Sample Lost Status is listed below:

```
struct SampleLostStatus {
  long total_count;
  long total_count_change;
};
```

The total_count value is the cumulative count of samples reported as lost. The total_count_change value is the incremental count of lost samples since the last time this status was accessed.

4.2.3.7 Subscription Matched Status

The SUBSCRIPTION_MATCHED status indicates that either a compatible data writer has been matched or a previously matched data writer has ceased to be matched. The IDL associated with the Subscription Matched Status is listed below:

```
struct SubscriptionMatchedStatus {
  long total_count;
  long total_count_change;
  long current_count;
  long current_count_change;
  InstanceHandle_t last_publication_handle;
};
```

The total_count value is the cumulative count of data writers that have compatibly matched this data reader. The total_count_change value is the incremental change in the total count since the last time this status was accessed. The current_count value is the current number of data writers matched to this data reader. The current_count_change value is the change in the current count since the last time this status was accessed. The last_publication_handle value is a handle for the last data writer matched.

4.2.4 Data Writer Status Types

4.2.4.1 Liveliness Lost Status

The LIVELINESS_LOST status indicates that the liveliness that the data writer committed through its Liveliness QoS has not been respected. This means that any connected data

readers will consider this data writer no longer active.The IDL associated with the Liveliness Lost Status is listed below:

```
struct LivelinessLostStatus {
  long total_count;
  long total_count_change;
};
```

The `total_count` value is the cumulative count of times that an alive data writer has become not alive. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed.

4.2.4.2 Offered Deadline Missed Status

The `OFFERED_DEADLINE_MISSED` status indicates that the deadline offered by the data writer has been missed for one or more instances. The IDL associated with the Offered Deadline Missed Status is listed below:

```
struct OfferedDeadlineMissedStatus {
  long total_count;
  long total_count_change;
  InstanceHandle_t last_instance_handle;
};
```

The `total_count` value is the cumulative count of times that deadlines have been missed for an instance. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed. The `last_instance_handle` value indicates the last instance that has missed a deadline.

4.2.4.3 Offered Incompatible QoS Status

The `OFFERED_INCOMPATIBLE_QOS` status indicates that an offered QoS was incompatible with the requested QoS of a data reader. The IDL associated with the Offered Incompatible QoS Status is listed below:

```
struct QosPolicyCount {
  QosPolicyId_t policy_id;
  long count;
};
typedef sequence<QosPolicyCount> QosPolicyCountSeq;

struct OfferedIncompatibleQosStatus {
  long total_count;
  long total_count_change;
  QosPolicyId_t last_policy_id;
  QosPolicyCountSeq policies;
};
```

The `total_count` value is the cumulative count of times that data readers with incompatible QoS have been found. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed. The `last_policy_id` value identifies one

of the QoS policies that was incompatible in the last incompatibility detected. The `policies` value is a sequence of values that indicates the total number of incompatibilities that have been detected for each QoS policy.

4.2.4.4 Publication Matched Status

The `PUBLICATION_MATCHED` status indicates that either a compatible data reader has been matched or a previously matched data reader has ceased to be matched. The IDL associated with the Publication Matched Status is listed below:

```
struct PublicationMatchedStatus {
  long total_count;
  long total_count_change;
  long current_count;
  long current_count_change;
  InstanceHandle_t last_subscription_handle;
};
```

The `total_count` value is the cumulative count of data readers that have compatibly matched this data writer. The `total_count_change` value is the incremental change in the total count since the last time this status was accessed. The `current_count` value is the current number of data readers matched to this data writer. The `current_count_change` value is the change in the current count since the last time this status was accessed. The `last_subscription_handle` value is a handle for the last data reader matched.

4.3 Listeners

Each entity defines its own listener interface based on the statuses it can report. Any entity's listener interface also inherits from the listeners of its owned entities, allowing it to handle statuses for owned entities as well. For example, a subscriber listener directly defines an operation to handle Data On Readers statuses and inherits from the data reader listener as well.

Each status operation takes the general form of `on_<status_name>(<entity>, <status_struct>)`, where `<status_name>` is the name of the status being reported, `<entity>` is a reference to the entity the status is reported for, and `<status_struct>` is the structure with details of the status. Read statuses omit the second parameter. For example, here is the operation for the Sample Lost status:

```
  void on_sample_lost(in DataReader the_reader, in SampleLostStatus status);
```

Listeners can either be passed to the factory function used to create their entity or explicitly set by calling `set_listener()` on the entity after it is created. Both of these functions also take a status mask as a parameter. The mask indicates which statuses are enabled in that listener. Mask bit values for each status are defined in DdsDcpsInfrastructure.idl:

```
module DDS {
  typedef unsigned long StatusKind;
  typedef unsigned long StatusMask; // bit-mask StatusKind

  const StatusKind INCONSISTENT_TOPIC_STATUS        = 0x0001 << 0;
  const StatusKind OFFERED_DEADLINE_MISSED_STATUS   = 0x0001 << 1;
  const StatusKind REQUESTED_DEADLINE_MISSED_STATUS = 0x0001 << 2;
  const StatusKind OFFERED_INCOMPATIBLE_QOS_STATUS  = 0x0001 << 5;
  const StatusKind REQUESTED_INCOMPATIBLE_QOS_STATUS= 0x0001 << 6;
  const StatusKind SAMPLE_LOST_STATUS               = 0x0001 << 7;
  const StatusKind SAMPLE_REJECTED_STATUS           = 0x0001 << 8;
  const StatusKind DATA_ON_READERS_STATUS           = 0x0001 << 9;
  const StatusKind DATA_AVAILABLE_STATUS            = 0x0001 << 10;
  const StatusKind LIVELINESS_LOST_STATUS           = 0x0001 << 11;
  const StatusKind LIVELINESS_CHANGED_STATUS        = 0x0001 << 12;
  const StatusKind PUBLICATION_MATCHED_STATUS       = 0x0001 << 13;
  const StatusKind SUBSCRIPTION_MATCHED_STATUS      = 0x0001 << 14;
};
```

Simply do a bit-wise “or” of the desired status bits to construct a mask for your listener. Here is an example of attaching a listener to a data reader (for just Data Available statuses):

```
    DDS::DataReaderListener_var listener (new DataReaderListenerImpl);
    // Create the Datareader
    DDS::DataReader_var dr = sub->create_datareader(
      topic,
      DATAREADER_QOS_DEFAULT,
      listener,
      DDS::DATA_AVAILABLE_STATUS);
```

Here is an example showing how to change the listener using `set_listener()`:

```
  dr->set_listener(listener,
                   DDS::DATA_AVAILABLE_STATUS |
                   DDS::LIVELINESS_CHANGED_STATUS);
```

When a plain communication status changes, OpenDDS invokes the most specific relevant listener operation. This means, for example, that a data reader’s listener would take precedence over the subscriber’s listener for statuses related to the data reader.

The following sections define the different listener interfaces. For more details on the individual statuses, see 4.2 .

4.3.1 Topic Listener

```
interface TopicListener : Listener {
  void on_inconsistent_topic(in Topic the_topic,
                             in InconsistentTopicStatus status);
};
```

4.3.2 Data Writer Listener

```
interface DataWriterListener : Listener {
  void on_offered_deadline_missed(in DataWriter writer,
```

```
                                    in OfferedDeadlineMissedStatus status);
  void on_offered_incompatible_qos(in DataWriter writer,
                                    in OfferedIncompatibleQosStatus status);
  void on_liveliness_lost(in DataWriter writer,
                          in LivelinessLostStatus status);
  void on_publication_matched(in DataWriter writer,
                              in PublicationMatchedStatus status);
};
```

4.3.3 Publisher Listener

```
interface PublisherListener : DataWriterListener {
};
```

4.3.4 Data Reader Listener

```
interface DataReaderListener : Listener {
  void on_requested_deadline_missed(in DataReader the_reader,
                                    in RequestedDeadlineMissedStatus status);
  void on_requested_incompatible_qos(in DataReader the_reader,
                                     in RequestedIncompatibleQosStatus status);
  void on_sample_rejected(in DataReader the_reader,
                          in SampleRejectedStatus status);
  void on_liveliness_changed(in DataReader the_reader,
                             in LivelinessChangedStatus status);
  void on_data_available(in DataReader the_reader);
  void on_subscription_matched(in DataReader the_reader,
                               in SubscriptionMatchedStatus status);
  void on_sample_lost(in DataReader the_reader,
                      in SampleLostStatus status);
};
```

4.3.5 Subscriber Listener

```
interface SubscriberListener : DataReaderListener {
  void on_data_on_readers(in Subscriber the_subscriber);
};
```

4.3.6 Domain Participant Listener

```
interface DomainParticipantListener : TopicListener,
                                      PublisherListener,
                                      SubscriberListener {
};
```

4.4 Conditions

The DDS specification defines four types of condition:

- Status Condition

- Read Condition
- Query Condition
- Guard Condition

4.4.1 Status Condition

Each entity has a status condition object associated with it and a `get_statuscondition()` operation that lets applications access the status condition. Each condition has a set of enabled statuses that can trigger that condition. Attaching one or more conditions to a wait set allows application developers to wait on the condition's status set. Once an enabled status is triggered, the wait call returns from the wait set and the developer can query the relevant status condition on the entity. Querying the status condition resets the status.

4.4.1.1 Status Condition Example

This example enables the Offered Incompatible QoS status on a data writer, waits for it, and then queries it when it triggers. The first step is to get the status condition from the data writer, enable the desired status, and attach it to a wait set:

```
DDS::StatusCondition_var cond = data_writer->get_statuscondition();
cond->set_enabled_statuses(DDS::OFFERED_INCOMPATIBLE_QOS_STATUS);

DDS::WaitSet_var ws = new DDS::WaitSet;
ws->attach_condition(cond);
```

Now we can wait ten seconds for the condition:

```
DDS::ConditionSeq active;
DDS::Duration ten_seconds = {10, 0};
int result = ws->wait(active, ten_seconds);
```

The result of this operation is either a timeout or a set of triggered conditions in the active sequence:

```
if (result == DDS::RETCODE_TIMEOUT) {
  cout << "Wait timed out" << std::endl;
} else if (result == DDS::RETCODE_OK) {
  DDS::OfferedIncompatibleQosStatus incompatibleStatus;
  data_writer->get_offered_incompatible_qos(incompatibleStatus);
  // Access status fields as desired...
}
```

Developers have the option of attaching multiple conditions to a single wait set as well as enabling multiple statuses per condition.

4.4.2 Additional Condition Types

The DDS specification also defines three other types of conditions: read conditions, query conditions, and guard conditions. These conditions do not directly involve the processing of

statuses but allow the integration of other activities into the condition and wait set mechanisms. These are other conditions are briefly described here. For more information see the DDS specification or the OpenDDS tests in $DDS_ROOT/tests/.

4.4.2.1 Read Conditions

Read conditions are created using the data reader and the same masks that are passed to the read and take operations. When waiting on this condition, it is triggered whenever samples match the specified masks. Those samples can then be retrieved using the `read_w_condition()` and `take_w_condition()` operations which take the read condition as a parameter.

4.4.2.2 Query Conditions

Query conditions are a specialized form of read conditions that are created with a limited form of an SQL-like query. This allows applications to filter the data samples that trigger the condition and then are read use the normal read condition mechanisms. See Section 5.3 for more information about query conditions.

4.4.2.3 Guard Conditions

The guard condition is a simple interface that allows the application to create its own condition object and trigger it when application events (external to OpenDDS) occur.

CHAPTER 5

Content-Subscription Profile

5.1 Introduction

The Content-Subscription Profile of DDS consists of three features which enable a data reader's behavior to be influenced by the content of the data samples it receives. These three features are:

- Content-Filtered Topic
- Query Condition
- Multi Topic

The content-filtered topic and multi topic interfaces inherit from the `TopicDescription` interface (and not from the `Topic` interface, as the names may suggest).

Content-filtered topic and query condition allow filtering (selection) of data samples using a SQL-like parameterized query string. Additionally, query condition allows sorting the result set returned from a data reader's `read()` or `take()` operation. Multi topic also has this selection capability as well as the ability to aggregate data from different data writers into a single data type and data reader.

If you are not planning on using the Content-Subscription Profile features in your application, you can configure OpenDDS to remove support for it at build time. See page 13 for information on disabling this support.

5.2 Content-Filtered Topic

The domain participant interface contains operations for creating and deleting a content-filtered topic. Creating a content-filtered topic requires the following parameters:

- Name
  Assigns a name to this content-filtered topic which could later be used with the `lookup_topicdescription()` operation.
- Related topic
  Specifies the topic that this content-filtered topic is based on. This is the same topic that matched data writers will use to publish data samples.
- Filter expression
  An SQL-like expression (see section 5.2.1) which defines the subset of samples published on the related topic that should be received by the content-filtered topic's data readers.
- Expression parameters
  The filter expression can contain parameter placeholders. This argument provides initial values for those parameters. The expression parameters can be changed after the content-filtered topic is created (the filter expression cannot be changed).

Once the content-filtered topic has been created, it is used by the subscriber's `create_datareader()` operation to obtain a content-filtering data reader. This data reader is functionally equivalent to a normal data reader except that incoming data samples which do not meet the filter expression's criteria are dropped.

Filter expressions are first evaluated at the publisher so that data samples which would be ignored by the subscriber can be dropped before even getting to the transport. This feature can be turned off with `-DCPSPublisherContentFilter 0` or the equivalent setting in the `[common]` section of the configuration file. The behavior of non-default `DEADLINE` or `LIVELINESS` QoS policies may be affected by this policy. Special consideration must be given to how the "missing" samples impact the QoS behavior, see the document in `docs/design/CONTENT_SUBSCRIPTION`.

Note *RTPS transport does not always do Writer-side filtering. It does not currently implement transport level filtering, but may be able to filter above the transport layer.*

5.2.1 Filter Expressions

The formal grammar for filter expressions is defined in Annex A of the DDS specification. This section provides an informal summary of that grammar. Query expressions (5.3.1) and topic expressions (5.4.1) are also defined in Annex A.

Filter expressions are combinations of one or more predicates. Each predicate is a logical expression taking one of two forms:

- `<arg1> <RelOp> <arg2>`
  - `arg1` and `arg2` are arguments which may be either a literal value (integer, character, floating-point, string, or enumeration), a parameter placeholder of the form `%n` (where n is a zero-based index into the parameter sequence), or a field reference.
  - At least one of the arguments must be a field reference, which is the name of an IDL struct field, optionally followed by any number of '.' and another field name to represent nested structures.
  - `RelOp` is a relational operator from the list: `=`, `>`, `>=`, `<`, `<=`, `<>`, and '`like`'. '`like`' is a wildcard match using `%` to match any number of characters and _ to match a single character.
  - Examples of this form of predicate include: `a = 'z'`, `b <> 'str'`, `c < d`, `e = 'enumerator'`, `f >= 3.14e3`, `27 > g`, `h <> i`, `j.k.l like %0`
- <arg1> [NOT] BETWEEN <arg2> AND <arg3>
  - In this form, argument 1 must be a field reference and arguments 2 and 3 must each be a literal value or parameter placeholder.

Any number of predicates can be combined through the use of parenthesis and the Boolean operators `AND`, `OR`, and `NOT` to form a filter expression.

5.2.2 Content-Filtered Topic Example

The code snippet below creates a content-filtered topic for the Message type. First, here is the IDL for Message:

```
module Messenger {
#pragma DCPS_DATA_TYPE "Messenger::Message"
  struct Message {
    long id;
  };
};
```

Next we have the code that creates the data reader:

```
CORBA::String_var type_name = message_type_support->get_type_name();
DDS::Topic_var topic = dp->create_topic("MyTopic",
                                        type_name,
                                        TOPIC_QOS_DEFAULT,
                                        NULL,
                                        OpenDDS::DCPS::DEFAULT_STATUS_MASK);
DDS::ContentFilteredTopic_var cft =
  participant->create_contentfilteredtopic("MyTopic-Filtered",
                                           topic,
                                           "id > 1",
```

```
                                                StringSeq());
  DDS::DataReader_var dr =
    subscriber->create_datareader(cft,
                                  dr_qos,
                                  NULL,
                                  OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

The data reader ‘dr’ will only receive samples that have values of ‘id’ greater than 1.

5.3 Query Condition

The query condition interface inherits from the read condition interface, therefore query conditions have all of the capabilities of read conditions along with the additional capabilities described in this section. One of those inherited capabilities is that the query condition can be used like any other condition with a wait set (see Section 4.4).

The DataReader interface contains operations for creating (create_querycondition) and deleting (delete_readcondition) a query condition. Creating a query condition requires the following parameters:

- Sample, view, and instance state masks
  These are the same state masks that would be passed to create_readcondition(), read(), or take().
- Query expression
  An SQL-like expression (see 5.3.1) describing a subset of samples which cause the condition to be triggered. This same expression is used to filter the data set returned from a read_w_condition() or take_w_condition() operation. It may also impose a sort order (ORDER BY) on that data set.
- Query parameters
  The query expression can contain parameter placeholders. This argument provides initial values for those parameters. The query parameters can be changed after the query condition is created (the query expression cannot be changed).

A particular query condition can be used with a wait set (attach_condition), with a data reader (read_w_condition, take_w_condition, read_next_instance_w_condition, take_next_instance_w_condition), or both. When used with a wait set, the ORDER BY clause has no effect on triggering the wait set. When used with a data reader’s read*() or take*() operation, the resulting data set will only contain samples which match the query expression and they will be ordered by the ORDER BY fields, if an ORDER BY clause is present.

5.3.1 Query Expressions

Query expressions are a superset of filter expressions (see section 5.2.1). Following the filter expression, the query expression can optionally have an ORDER BY keyword followed by

a comma-separated list of field references. If the `ORDER BY` clause is present, the filter expression may be empty. The following strings are examples of query expressions:

- m > 100 ORDER BY n
- ORDER BY p.q, r, s.t.u
- NOT v LIKE 'z%'

5.3.2 Query Condition Example

The following code snippet creates and uses a query condition for a type that uses struct ‘Message’ with field ‘key’ (an integral type).

```
DDS::QueryCondition_var dr_qc =
  dr->create_querycondition(DDS::ANY_SAMPLE_STATE,
                            DDS::ANY_VIEW_STATE,
                            DDS::ALIVE_INSTANCE_STATE,
                            "key > 1",
                            DDS::StringSeq());
DDS::WaitSet_var ws = new DDS::WaitSet;
ws->attach_condition(dr_qc);
DDS::ConditionSeq active;
DDS::Duration_t three_sec = {3, 0};
DDS::ReturnCode_t ret = ws->wait(active, three_sec);
  // error handling not shown
ws->detach_condition(dr_qc);
MessageDataReader_var mdr = MessageDataReader::_narrow(dr);
MessageSeq data;
DDS::SampleInfoSeq infoseq;
ret = mdr->take_w_condition(data, infoseq, DDS::LENGTH_UNLIMITED, dr_qc);
  // error handling not shown
dr->delete_readcondition(dr_qc);
```

Any sample received with `key <= 1` would neither trigger the condition (to satisfy the wait) nor be returned in the ‘data’ sequence from `take_w_condition()`.

5.4 Multi Topic

Multi topic is a more complex feature than the other two Content-Subscription features, therefore describing it requires some new terminology.

The `MultiTopic` interface inherits from the `TopicDescription` interface, just like `ContentFilteredTopic` does. A data reader created for the multi topic is known as a “multi topic data reader.” A multi topic data reader receives samples belonging to any number of regular topics. These topics are known as its “constituent topics.” The multi topic has a DCPS data type known as the “resulting type.” The multi topic data reader implements the type-specific data reader interface for the resulting type. For example, if the resulting type is Message, then the multi topic data reader can be narrowed to the `MessageDataReader` interface.

The multi topic’s topic expression (see section 5.4.1) describes how the distinct fields of the incoming data (on the constituent topics) are mapped to the fields of the resulting type.

The domain participant interface contains operations for creating and deleting a multi topic. Creating a multi topic requires the following parameters:

- Name
  Assigns a name to this multi topic which could later be used with the `lookup_topicdescription()` operation.
- Type name
  Specifies the resulting type of the multi topic. This type must have its type support registered before creating the multi topic.
- Topic expression (also known as subscription expression)
  An SQL-like expression (see section 5.4.1) which defines the mapping of constituent topic fields to resulting type fields. It can also specify a filter (`WHERE` clause).
- Expression parameters
  The topic expression can contain parameter placeholders. This argument provides initial values for those parameters. The expression parameters can be changed after the multi topic is created (the topic expression cannot be changed).

Once the multi topic has been created, it is used by the subscriber's `create_datareader()` operation to obtain a multi topic data reader. This data reader is used by the application to receive the constructed samples of the resulting type. The manner in which these samples are constructed is described below in section 5.4.2.2 .

5.4.1 Topic Expressions

Topic expressions use a syntax that is very similar to a complete SQL query:

```
SELECT <aggregation> FROM <selection> [WHERE <condition>]
```

- The aggregation can be either a "*" or a comma separated list of field specifiers. Each field specifier has the following syntax:
  - <constituent_field> [[AS] <resulting_field>]]
  - `constituent_field` is a field reference (see section 1.1.1) to a field in one of the constituent topics (which topic is not specified).
  - The optional resulting_field is a field reference to a field in the resulting type. If present, the `resulting_field` is the destination for the constituent_field in the constructed sample. If absent, the `constituent_field` data is assigned to a field with the same name in the resulting type. The optional "`AS`" has no effect.
  - If a "*" is used as the aggregation, each field in the resulting type is assigned the value from a same-named field in one of the constituent topic types.

- The selection lists one or more constituent topic names. Topic names are separated by a "join" keyword (all 3 join keywords are equivalent):
  - <topic> [{NATURAL INNER | NATURAL | INNER NATURAL} JOIN <topic>]...
  - Topic names must contain only letters, digits, and dashes (but may not start with a digit).
  - The natural join operation is commutative and associative, thus the order of topics has no impact.
  - The semantics of the natural join are that any fields with the same name are treated as "join keys" for the purpose of combining data from the topics in which those keys appear. The join operation is described in more detail in the subsequent sections of this chapter.
- The condition has the exact same syntax and semantics as the filter expression (see section 5.2.1). Field references in the condition must match field names in the resulting types, not field names in the constituent topic types.

5.4.2 Usage Notes

5.4.2.1 Join Keys and DCPS Data Keys

The concept of DCPS data keys (`#pragma DCPS_DATA_KEY`) has already been discussed in Section 2.1.1. Join keys for the multi topic are a distinct but related concept.

A join key is any field name that occurs in the struct for more than one constituent topic. The existence of the join key enforces a constraint on how data samples of those topics are combined into a constructed sample (see section 5.4.2.2). Specifically, the value of that key must be equal for those data samples from the constituent topics to be combined into a sample of the resulting type. If multiple join keys are common to the same two or more topics, the values of all keys must be equal in order for the data to be combined.

The DDS specification requires that join key fields have the same type. Additionally, OpenDDS imposes two requirements on how the IDL must define DCPS data keys to work with multi topics:

1) Each join key field must also be a DCPS data key for the types of its constituent topics.
2) The resulting type must contain each of the join keys, and those fields must be DCPS data keys for the resulting type.

The example in section 5.4.3.1 meets both of these requirements. Note that it is not necessary to list the join keys in the aggregation (SELECT clause).

5.4.2.2 How Resulting Samples are Constructed

Although many concepts in multi topic are borrowed from the domain of relational databases, a real-time middleware such as DDS is not a database. Instead of processing a batch of data at a time, each sample arriving at the data reader from one of the constituent topics triggers multi-topic-specific processing that results in the construction of zero, one, or many samples of the resulting type and insertion of those constructed samples into the multi topic data reader.

Specifically, the arrival of a sample on constituent topic "A" with type "TA" results in the following steps in the multi topic data reader (this is a simplification of the actual algorithm):

1) A sample of the resulting type is constructed, and fields from TA which exist in the resulting type and are in the aggregation (or are join keys) are copied from the incoming sample to the constructed sample.

2) Each topic "B" which has at least one join key in common with A is considered for a join operation. The join reads READ_SAMPLE_STATE samples on topic B with key values matching those in the constructed sample. The result of the join may be zero, one, or many samples. Fields from TB are copied to the resulting sample as described in step 1.

3) Join keys of topic "B" (connecting it to other topics) are then processed as described in step 2, and this continues to all other topics that are connected by join keys.

4) Any constituent topics that were not visited in steps 2 or 3 are processed as "cross joins" (also known as cross-product joins). These are joins with no key constraints.

5) If any constructed samples result, they are inserted into the multi topic data reader's internal data structures as if they had arrived via the normal mechanisms. Application listeners and conditions are notified.

5.4.2.3 Use with Subscriber Listeners

If the application has registered a subscriber listener for read condition status changes (DATA_ON_READERS_STATUS) with the same subscriber that also contains a multi topic, then the application must invoke notify_datareaders() in its implementation of the subscriber listener's on_data_on_readers() callback method. This requirement is necessary because the multi topic internally uses data reader listeners, which are preempted when a subscriber listener is registered.

5.4.3 Multi Topic Example

This example is based on the example topic expression used in Annex A section A.3 of the DDS specification. It illustrates how the properties of the multi topic join operation can be used to correlate data from separate topics (and possibly distinct publishers).

5.4.3.1 IDL and Topic Expression

Often times we will use the same string as both the topic name and topic type. In this example we will use distinct strings for the type names and topic names, in order to illustrate when each is used.

Here is the IDL for the constituent topic data types:

```
#pragma DCPS_DATA_TYPE "LocationInfo"
#pragma DCPS_DATA_KEY "LocationInfo flight_id"
struct LocationInfo {
  unsigned long flight_id;
  long x;
  long y;
  long z;
};

#pragma DCPS_DATA_TYPE "PlanInfo"
#pragma DCPS_DATA_KEY "PlanInfo flight_id"
struct PlanInfo {
  unsigned long flight_id;
  string flight_name;
  string tailno;
};
```

Note that the names and types of the key fields match, so they are designed to be used as join keys. The resulting type (below) also has that key field.

Next we have the IDL for the resulting data type:

```
#pragma DCPS_DATA_TYPE "Resulting"
#pragma DCPS_DATA_KEY "Resulting flight_id"
struct Resulting {
  unsigned long flight_id;
  string flight_name;
  long x;
  long y;
  long height;
};
```

Based on this IDL, the following topic expression can be used to combine data from a topic *`Location`* which uses type `LocationInfo` and a topic *`FlightPlan`* which uses type `PlanInfo`:

```
SELECT flight_name, x, y, z AS height FROM Location NATURAL JOIN FlightPlan WHERE height
< 1000 AND x <23
```

Taken together, the IDL and the topic expression describe how this multi topic will work. The multi topic data reader will construct samples which belong to instances keyed by `flight_id`. The instance of the resulting type will only come into existence once the corresponding instances are available from both the *`Location`* and *`FlightPlan`* topics. Some

other domain participant or participants within the domain will publish data on those topics, and they don't even need to be aware of one another. Since they each use the same `flight_id` to refer to flights, the multi topic can correlate the incoming data from disparate sources.

5.4.3.2 Creating the Multi Topic Data Reader

Creating a data reader for the multi topic consists of a few steps. First the type support for the resulting type is registered, then the multi topic itself is created, followed by the data reader:

```
ResultingTypeSupport_var ts_res = new ResultingTypeSupportImpl;
ts_res->register_type(dp, "");
CORBA::String_var type_name = ts_res->get_type_name();
DDS::MultiTopic_var mt =
  dp->create_multitopic("MyMultiTopic",
                        type_name,
                        "SELECT flight_name, x, y, z AS height "
                          "FROM Location NATURAL JOIN FlightPlan "
                          "WHERE height < 1000 AND x<23",
                        DDS::StringSeq());
DDS::DataReader_var dr =
  sub->create_datareader(mt,
                         DATAREADER_QOS_DEFAULT,
                         NULL,
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

5.4.3.3 Reading Data with the Multi Topic Data Reader

From an API perspective, the multi topic data reader is identical to any other typed data reader for the resulting type. This example uses a wait set and a read condition in order to block until data is available.

```
DDS::WaitSet_var ws = new DDS::WaitSet;
DDS::ReadCondition_var rc =
  dr->create_readcondition(DDS::ANY_SAMPLE_STATE,
                           DDS::ANY_VIEW_STATE,
                           DDS::ANY_INSTANCE_STATE);
ws->attach_condition(rc);
DDS::Duration_t infinite = {DDS::DURATION_INFINITE_SEC,
                            DDS::DURATION_INFINITE_NSEC};
DDS::ConditionSeq active;
ws->wait(active, infinite); // error handling not shown
ws->detach_condition(rc);
ResultingDataReader_var res_dr = ResultingDataReader::_narrow(dr);
ResultingSeq data;
DDS::SampleInfoSeq info;
res_dr->take_w_condition(data, info, DDS::LENGTH_UNLIMITED, rc);
```

CHAPTER 6

Built-In Topics

6.1 Introduction

In OpenDDS, Built-In-Topics are created and published by default to exchange information about DDS participants operating in the deployment. When OpenDDS is used in a centralized discovery approach using the `DCPSInfoRepo` service, the Built-In-Topics are published by this service. For DDS-RTPS deployments, the internal OpenDDS implementation instantiated in a process creates and uses Built-In-Topics to accomplish similar exchange of discovery information with other DDS participants. See Section 7.3.3 for a description of RTPS discovery configuration.

6.2 Built-In Topics for DCPSInfoRepo Configuration

When using the `DCPSInfoRepo` a command line option of `-NOBITS` may be used to suppress publication of built-in topics.

Note *An RTPS discovery configuration must use Built-In-Topics, so this should not be disabled.*

Four separate topics are defined for each domain . Each is dedicated to a particular entity (domain participant, topic, data writer, data reader) and publishes instances describing the state for each entity in the domain.

Subscriptions to built-in topics are automatically created for each domain participant. A participant's support for Built-In-Topics can be toggled via the `DCPSBit` configuration option (see the table in Section 7.2) (Note: this option cannot be used for RTPS discovery). To view the built-in topic data, simply obtain the built-in Subscriber and then use it to access the Data Reader for the built-in topic of interest. The Data Reader can then be used like any other Data Reader.

Sections 6.4 through 6.7 provide details on the data published for each of the four built-in topics. An example showing how to read from a built-in topic follows those sections.

If you are not planning on using Built-in-Topics in your application, you can configure OpenDDS to remove Built-In-Topic support at build time. Doing so can reduce the footprint of the core DDS library by up to 30%. See Section 1.3.2 for information on disabling Built-In-Topic support.

6.3 Building Without Built-In Topic Support

6.4 DCPSParticipant Topic

The `DCPSParticipant` topic publishes information about the Domain Participants of the Domain. Here is the IDL that defines the structure published for this topic:

```
struct ParticipantBuiltinTopicData {
  BuiltinTopicKey_t key;  // struct containing an array of 3 longs
  UserDataQosPolicy user_data;
};
```

Each Domain Participant is defined by a unique key and is its own instance within this topic.

6.5 DCPSTopic Topic

Note *OpenDDS does not support this Built-In-Topic when configured for RTPS discovery.*

The `DCPSTopic` topic publishes information about the topics in the domain. Here is the IDL that defines the structure published for this topic:

```
struct TopicBuiltinTopicData {
  BuiltinTopicKey_t key;
  string name;
  string type_name;
```

```
  DurabilityQosPolicy durability;
  QosPolicy deadline;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy liveliness;
  ReliabilityQosPolicy reliability;
  TransportPriorityQosPolicy transport_priority;
  LifespanQosPolicy lifespan;
  DestinationOrderQosPolicy destination_order;
  HistoryQosPolicy history;
  ResourceLimitsQosPolicy resource_limits;
  OwnershipQosPolicy ownership;
  TopicDataQosPolicy topic_data;
};
```

Each topic is identified by a unique key and is its own instance within this built-in topic. The members above identify the name of the topic, the name of the topic type, and the set of QoS policies for that topic.

6.6 DCPSPublication Topic

The `DCPSPublication` topic publishes information about the Data Writers in the Domain. Here is the IDL that defines the structure published for this topic:

```
struct PublicationBuiltinTopicData {
  BuiltinTopicKey_t key;
  BuiltinTopicKey_t participant_key;
  string topic_name;
  string type_name;
  DurabilityQosPolicy durability;
  DeadlineQosPolicy deadline;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy liveliness;
  ReliabilityQosPolicy reliability;
  LifespanQosPolicy lifespan;
  UserDataQosPolicy user_data;
  OwnershipStrengthQosPolicy ownership_strength;
  PresentationQosPolicy presentation;
  PartitionQosPolicy partition;
  TopicDataQosPolicy topic_data;
  GroupDataQosPolicy group_data;
};
```

Each Data Writer is assigned a unique key when it is created and defines its own instance within this topic. The fields above identify the Domain Participant (via its key) that the Data Writer belongs to, the topic name and type, and the various QoS policies applied to the Data Writer.

6.7 DCPSSubscription Topic

The `DCPSSubscription` topic publishes information about the Data Readers in the Domain. Here is the IDL that defines the structure published for this topic:

```
struct SubscriptionBuiltinTopicData {
  BuiltinTopicKey_t key;
```

```
  BuiltinTopicKey_t participant_key;
  string topic_name;
  string type_name;
  DurabilityQosPolicy durability;
  DeadlineQosPolicy deadline;
  LatencyBudgetQosPolicy latency_budget;
  LivelinessQosPolicy liveliness;
  ReliabilityQosPolicy reliability;
  DestinationOrderQosPolicy destination_order;
  UserDataQosPolicy user_data;
  TimeBasedFilterQosPolicy time_based_filter;
  PresentationQosPolicy presentation;
  PartitionQosPolicy partition;
  TopicDataQosPolicy topic_data;
  GroupDataQosPolicy group_data;
};
```

Each Data Reader is assigned a unique key when it is created and defines its own instance within this topic. The fields above identify the Domain Participant (via its key) that the Data Reader belongs to, the topic name and type, and the various QoS policies applied to the Data Reader.

6.8 Built-In Topic Subscription Example

The following code uses a domain participant to get the built-in subscriber. It then uses the subscriber to get the Data Reader for the `DCPSParticipant` topic and subsequently reads samples for that reader.

```
Subscriber_var bit_subscriber = participant->get_builtin_subscriber() ;
DDS::DataReader_var dr =
  bit_subscriber->lookup_datareader(BUILT_IN_PARTICIPANT_TOPIC);
DDS::ParticipantBuiltinTopicDataDataReader_var part_dr =
  DDS::ParticipantBuiltinTopicDataDataReader::_narrow(dr);

DDS::ParticipantBuiltinTopicDataSeq part_data;
DDS::SampleInfoSeq infos;
DDS::ReturnCode_t ret = part_dr->read ( part_data, infos, 20,
                                        DDS::ANY_SAMPLE_STATE,
                                        DDS::ANY_VIEW_STATE,
                                        DDS::ANY_INSTANCE_STATE) ;

// Check return status and read the participant data
```

The code for the other built-in topics is similar.

CHAPTER 7

Configuring OpenDDS

7.1 Configuration Approach

OpenDDS includes a file-based configuration framework for configuring both global options as well as discovery and transport configurations for publishers and subscribers. Additionally, OpenDDS allows for overriding some configuration file entries with command line options along with the ability to set configuration in code. This chapter summarizes the configuration options supported by OpenDDS.

An OpenDDS configuration fall into three main areas:

1) **Common Configuration Options** - these options allow the designer to tailor the behavioral aspects of DCPS entities more globally and also allow separately deployed processes in a computing environment to share common settings for the specified behavior (e.g. all data readers/data writers should use RTPS discovery)

2) **Discovery Configuration Options** - OpenDDS allows for multiple approaches to achieve data writer/data reader discovery as detailed in Section 7.3 . Configuration options discussed here demonstrate how to customize discovery to the needs of the DDS application environment.

3) **Transport Configuration Options** - the Extensible Transport Framework (ETF) abstracts the transport layer from the remainder of the OpenDDS functionality, so the pluggable transport selections can be configured separately.

A configuration file can have up to six different types of sections in it that relate to common, discovery, or transport related configuration options. Table 7-1 shows a list of the available configuration section types as they relate to the area of OpenDDS that is configured.

Table 7-1 Configuration File Sections

Configuration Focus Area	Configuration File Section Title
Common	`[common]`

Discovery	`[domain]`
	`[repository]`
	`[rtps_discovery]`
	`[common]`[3]

Transport	`[common]`
	`[config]`
	`[transport]`

For each of the section types with the exception of `[common]`, the syntax of a section header takes the form of `[section type/instance]`. For example, a `[repository]` section type would always be used in a configuration file like so:

`[repository/repo_1]` where `repository` is the section type and `repo_1` is an instance name of a repository configuration. How to use instances to configure discovery and transports is explained further in Sections 7.3 and 7.4 .

The `-DCPSConfigFile` command-line argument is used to pass the location of a configuration file into OpenDDS executables. For example:

Windows:

```
publisher -DCPSConfigFile pub.ini
```

Unix:

```
./publisher -DCPSConfigFile pub.ini
```

The above example causes the OpenDDS service participant to read configuration options from the pub.ini configuration file. More accurately, the publisher's command-line arguments are passed to the service participant singleton when we initialize the domain

3 When used for discovery, the `[common]` section is only used for basic discovery settings

participant factory. This is accomplished in the preceding examples by using the `TheParticipantFactoryWithArgs` macro:

```
#include <dds/DCPS/Service_Participant.h>

int main (int argc, char* argv[])
{

  DDS::DomainParticipantFactory_var dpf =
    TheParticipantFactoryWithArgs(argc, argv);
```

The `Service_Participant` class also provides methods that allow an application to configure the DDS service. See the header file *`$DDS_ROOT/dds/DCPS/Service_Participant.h`* for details.

The following subsections detail each of the configuration file sections and the available options related to those sections.

7.2 Common Configuration Options

The `[common]` section of the OpenDDS configuration file contains options such as debugging output, the default object reference of the `DCPSInfoRepo` process, and memory preallocation settings. A sample `[common]` section follows:

```
[common]
DCPSDebugLevel=0
DCPSInfoRepo=localhost:12345
DCPSLivelinessFactor=80
DCPSChunks=20
DCPSChunksAssociationMultiplier=10
DCPSBitTransportPort=
DCPSBitLookupDurationMsec=2000
DCPSPendingTimeout=30
```

It is not necessary to specify every option.

Option values in the `[common]` section with names that begin with "`DCPS`" can be overridden by a command-line argument. The command-line argument has the same name as the configuration option with a "`-`" prepended to it. For example:

```
subscriber -DCPSInfoRepo localhost:12345
```

The following table summarizes the `[common]` configuration options:

Table 7-2 Common Configuration Options

Option	Description	Default
`DCPSBit=[1|0]`	Toggle Built-In-Topic support.	1

Option	Description	Default
DCPSBitLookupDurationMsec =msec	The maximum duration in milliseconds that the framework will wait for latent Built-In Topic information when retrieving BIT data given an instance handle. The participant code may get an instance handle for a remote entity before the framework receives and processes the related BIT information. The framework waits for up to the given amount of time before it fails the operation.	2000
DCPSBitTransportIPAddress =addr	IP address identifying the local interface to be used by tcp transport for the Built-In Topics. **NOTE**: This property is only applicable to a DCPSInfoRepo configuration.	INADDR_ANY
DCPSBitTransportPort=port	Port used by the tcp transport for Built-In Topics.If the default of '0' is used, the operating system will choose a port to use. **NOTE**: This property is only applicable to a DCPSInfoRepo configuration.	0
DCPSChunks=n	Configurable number of chunks that a data writer's and reader's cached allocators will preallocate when the RESOURCE_LIMITS QoS value is infinite. When all of the preallocated chunks are in use, OpenDDS allocates from the heap.	20
DCPSChunkAssociationMultiplier =n	Multiplier for the DCPSChunks or resource_limits.max_samples value to determine the total number of shallow copy chunks that are preallocated. Set this to a value greater than the number of connections so the preallocated chunk handles do not run out. A sample written to multiple data readers will not be copied multiple times but there is a shallow copy handle to that sample used to manage the delivery to each data reader. The size of the handle is small so there is not great need to set this value close to the number of connections.	10
DCPSDebugLevel=n	Integer value that controls the amount of debug information the DCPS layer prints. Valid values are 0 through 10.	0
0DCPSDefaultDiscovery= [DEFAULT_REPO |DEFAULT_RTPS |user-defined configuration instance name]	Specifies a discovery configuration to use for any domain not explicitly configured. DEFAULT_REPO translates to using the DCPSInfoRepo. DEFAULT_RTPS specifies the use of RTPS for discovery. See Section for details in configuring discovery.	DEFAULT_REPO

Option	Description	Default
`DCPSGlobalTransportConfig =name`	Specifies the name of the transport configuration that should be used as the global configuration. This configuration is used by all entities that do not otherwise specify a transport configuration. A special value of $file uses a transport configuration that includes all transport instances defined in the configuration file.	The default configuration is used as described in
`DCPSInfoRepo=objref`	Object reference for locating the DCPS Information Repository. This can either be a full CORBA IOR or a simple host:port string.	`file://repo.ior`
`DCPSLivelinessFactor=n`	Percent of the liveliness lease duration after which a liveliness message is sent. A value of 80 implies a 20% cushion of latency from the last detected heartbeat message.	`80`
`DCPSPendingTimeout=sec`	The maximum duration in seconds a data writer will block to allow unsent samples to drain on deletion. By default, this option blocks indefinitely.	`0`
`DCPSPersistentDataDir=path`	The path on the file system where durable data will be stored. If the directory does not exist it will be created automatically.	`OpenDDS-durable-data-dir`
`DCPSPublisherContentFilter =[1|0]`	Controls the filter expression evaluation policy for content filtered topics. When enabled (1), the publisher may drop any samples, before handing them off to the transport when these samples would have been ignored by the subscriber.	`1`
`DCPSTransportDebugLevel=n`	Integer value for controlling the transport logging granularity. Legal values span from 0 to 5.	`0`
`scheduler=[SCHED_RR |SCHED_FIFO |SCHED_OTHER]`	Selects the thread scheduler to use. Setting the scheduler to a value other than the default requires privileges on most systems. A value of `SCHED_RR`, `SCHED_FIFO`, or `SCHED_OTHER` can be set. `SCHED_OTHER` is the default scheduler on most systems; `SCHED_RR` is a round robin scheduling algorithm; and `SCHED_FIFO` allows each thread to run until it either blocks or completes before switching to a different thread.	SCHED_OTHER
`scheduler_slice=usec`	Some operating systems, such as SunOS, require a time slice value to be set when selecting schedulers other than the default. For those systems, this option can be used to set a value in microseconds.	none

The `DCPSInfoRepo` option's value is passed to `CORBA::ORB::string_to_object()` and can be any Object URL type understandable by TAO (file, IOR, corbaloc, corbaname). A simplified endpoint description of the form `<host>:<port>` is also accepted. It is equivalent to `corbaloc::<host>:<port>/DCPSInfoRepo`.

The `DCPSChunks` option allows application developers to tune the amount of memory preallocated when the `RESOURCE_LIMITS` are set to infinite. Once the allocated memory is exhausted, additional chunks are allocated/deallocated from the heap. This feature of

allocating from the heap when the preallocated memory is exhausted provides flexibility but performance will decrease when the preallocated memory is exhausted.

7.3 Discovery Configuration

In DDS implementations, participants are instantiated in application processes and must discover one another to begin the communication process. A DDS implementation uses the feature of domains to give context to the data being exchanged between DDS participants using the domain.When DDS applications are written, participants are assigned to a domain and need to ensure their configuration allows each participant to discovery the other participants in the same domain.

OpenDDS offers a centralized discovery method or a peer-to-peer discovery method to accomplish discovery of participants operating in domains. The centralized method uses a separate service called the `DCPSInfoRepo` service. The peer-to-peer method uses the DDS-RTPS discovery protocol standard to achieve non-centralized discovery. Applications using either method can use a multitude of configuration options to match the deployment needs of the DDS application. Each method uses default values if no configuration is supplied either by command line or through configuration files.

The following sections show how to add configuration details to a configuration for more advanced discovery capabilities. Deployments may need features such as using multiple `DCPSInfoRepo` services or have a need to use DDS-RTPS discovery to satisfy interoperability requirements. To accomplish this a combination of the `[domain]`, `[repository]`, and `[rtps_discovery]` sections can be used.

7.3.1 Domain Configuration

An OpenDDS configuration file uses the `[domain]` section type to configure one or more discovery domains with each domain pointing to a discovery configuration in the same file.

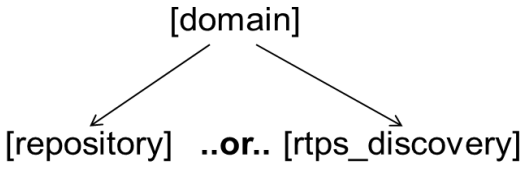


Figure 7-1: Only One Discovery Configuration Per Domain

OpenDDS applications can use a centralized discovery approach using the `DCPSInfoRepo` service or a peer-to-peer discovery approach using the RTPS discovery protocol standard or a combination of the two in the same deployment. The section type for the `DCPSInfoRepo` method is `[repository]` and the section type for an RTPS discovery configuration is `[rtps_discovery]`. A single domain can refer to only one type of discovery section.

See Sections 7.3.2 and 7.3.3 for more configuration options for `[repository]` and `[rtps_discovery]`.

Ultimately a domain is assigned an integer value and a configuration file can support this in two ways. The first is to simply make the instance value the integer value assigned to the domain as shown here:

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1
    (more properties...)
```

Our example configures a single domain identified by the domain keyword and followed by an instance value of `/1`. The instance value after the slash in this case is the integer value assigned to the domain. An alternative syntax for this same content is to use a more recognizable (friendly) name instead of a number for the domain name and then add the `DomainId` property to the section to give the integer value. Here is an example:

```
[domain/books]
DomainId=1
DiscoveryConfig=DiscoveryConfig1
```

The domain is given a friendly name of books. The `DomainId` property assigns the integer value of `1` needed by a DDS application reading the configuration. Multiple domain instances can be identified in a single configuration file in this format.

Once one or more domain instances are established, the discovery properties must be identified for that domain. In our example above, the property `DiscoveryConfig` must either point to another section that holds the discovery configuration or specify one of the internal default values for discovery (e.g. `DEFAULT_REPO` or `DEFAULT_RTPS`). The instance name in our example is `DiscoveryConfig1`. This instance name must be associated with a section type of either `[repository]` or `[rtps_discovery]`.

Here is an extension of our example:

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345
```

In this case our domain points to a `[repository]` section which is used for an OpenDDS `DCPSInfoRepo` service. See Section 7.3.2 for more details.

There are going to be occasions when specific domains are not identified in the configuration file. For example, if an OpenDDS application assigns a domain ID of 3 to its participants and the above example does not supply a configuration for domain id of 3 then the following can be used:

```
[common]
```

```
DCPSInfoRepo=host3.mydomain.com:12345
DCPSDefaultDiscovery=DEFAULT_REPO

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345
```

The DCPSDefaultDiscovery property tells the application to assign any participant that doesn't have a domain id found in the configuration file to use a discovery type of DEFAULT_REPO which means "use a DCPSInfoRepo service" and that DCPSInfoRepo service can be found at *host3.mydomain.com:12345*.

As shown in Table 7-2 the DCPSDefaultDiscovery property has two other values that can be used. The DEFAULT_RTPS constant value informs an application to tell any participant that doesn't have a domain configuration in the file to use RTPS discovery to discover other participants.

The final option for the DCPSDefaultDiscovery property is to tell an application to use one of the defined discovery configurations to be the default configuration for any participant domain that isn't called out in the file. Here is an example:

```
[common]
DCPSDefaultDiscovery=DiscoveryConfig2


[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository/DiscoveryConfig2]
RepositoryIor=host2.mydomain.com:12345
```

By adding the DCPSDefaultDiscovery property to the [common] section, any participant that hasn't been assigned to a domain id of 1 or 2 will use the configuration of DiscoveryConfig2. For more explanation of similar configuration for RTPS discovery see Section .

Here are the available properties for the [domain] section.

Table 7-3 Domain Section Configuration Properties

Section	Option	Value Description
[domain/[n | <NAME>]	DomainId=n	An integer value representing a Domain being associated with a repository.
	DomainRepoKey	Key value of the mapped repository (Deprecated. Provided for backward compatibility).

Section	Option	Value Description
	DiscoveryConfig= config instance name	A user-defined string that refers to the instance name of a [repository] or [rtps_discovery] section in the same configuration file or one of the internal default values (DEFAULT_REPO or DEFAULT_RTPS). (Also see the DCPSDefaultDiscovery property in Table 7-2)

7.3.2 Configuring Applications For DCPSInfoRepo

An OpenDDS DCPSInfoRepo is a service on a local or remote node used for participant discovery. Configuring how participants should find DCPSInfoRepo is the purpose of this section. Assume for example that the DCPSInfoRepo service is started on a host and port of *myhost.mydomain.com:12345*. Applications can make their OpenDDS participants aware of how to find this service through command line options or by reading a configuration file.

In our Getting Started example from 2.1.7, “Running the Example” the executables were given a command line parameter to find the DCPSInfoRepo service like so:

```
publisher -DCPSInfoRepo file://repo.ior
```

This assumes that the DCPSInfoRepo has been started with the following syntax:

Windows:

```
%DDS_ROOT%\bin\DCPSInfoRepo -o repo.ior
```

Unix:

```
$DDS_ROOT/bin/DCPSInfoRepo -o repo.ior
```

The DCPSInfoRepo service generates its location object information in this file and participants need to read this file to ultimately connect. The use of file based IORs to find a discovery service, however, is not practical in most production environments, so applications instead can use a command line option like the following to simply point to the host and port where the DCPSInfoRepo is running.

```
publisher -DCPSInfoRepo myhost.mydomain.com:12345
```

The above assumes that the DCPSInfoRepo has been started on a host (*myhost.mydomain.com*) as follows:

Windows:

```
%DDS_ROOT%\bin\DCPSInfoRepo -ORBListenEndpoints iiop://:12345
```

Unix:

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBListenEndpoints iiop://:12345
```

If an application needs to use a configuration file for other settings, it would become more convenient to place discovery content in the file and reduce command line complexity and clutter. The use of a configuration file also introduces the opportunity for multiple application processes to share common OpenDDS configuration. The above example can easily be moved to the `[common]` section of a configuration file (assume a file of *`pub.ini`*):

```
[common]
DCPSInfoRepo=myhost.mydomain.com:12345
```

The command line to start our executable would now change to the following:

```
publisher -DCSPConfigFile pub.ini
```

Reinforcing our example from the discussion of domains in section , a configuration file can specify domains with discovery configuration assigned to those domains. In this case the `RepositoryIor` property is used to take the same information that would be supplied on a command line to point to a running `DCPSInfoRepo` service. Two domains are configured here:

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=myhost.mydomain.com:12345

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository/DiscoveryConfig2]
RepositoryIor=host2.mydomain.com:12345
```

The `DiscoveryConfig` property under `[domain/1]` instructs all participants in domain `1` to use the configuration defined in an instance called `DiscoveryConfig1`. In the above, this is mapped to a `[repository]` section that gives the `RepositoryIor` value of *`myhost.mydomain.com:12345`*.

Finally, when configuring a `DCPSInfoRepo` the `DiscoveryConfig` property under a domain instance entry can also contain the value of `DEFAULT_REPO` which instructs a participant using this instance to use the definition of the property `DCPSInfoRepo` wherever it has been supplied. Consider the following configuration file as an example:

```
[common]
DCPSInfoRepo=localhost:12345

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=myhost.mydomain.com:12345

[domain/2]
DiscoveryConfig=DEFAULT_REPO
```

In this case any participant in domain 2 would be instructed to refer to the discovery property of DCPSInfoRepo, which is defined in the [common] section of our example. If the DCPSInfoRepo value is not supplied in the [common] section, it could alternatively be supplied as a parameter to the command line like so:

```
publisher -DCPSInfoRepo localhost:12345 -DCPSConfigFile pub.ini
```

This sets the value of DCPSInfoRepo such that if participants reading the configuration file pub.ini encounters DEFAULT_REPO, there is a value for it. If DCPSInfoRepo is not defined in a configuration file or on the command line, then the OpenDDS default value for DCPSInfoRepo is *file://repo.ior*. As mentioned prior, this is not likely to be the most useful in production environments and should lead to setting the value of DCPSInfoRepo by one of the means described in this section.

7.3.2.1 Configuring for Multiple DCPSInfoRepo Instances

The DDS entities in a single OpenDDS process can be associated with multiple DCPS information repositories (DCPSInfoRepo).

The repository information and domain associations can be configured using a configuration file, or via application API. Internal defaults, command line arguments, and configuration file options will work as-is for existing applications that do not want to use multiple DCPSInfoRepo associations.

Refer to Figure 7-2 as an example of a process that uses multiple DCPSInfoRepo repositories. Processes A and B are typical application processes that have been configured to communicate with one another and discover one another in InfoRepo_1. This is a simple use of basic discovery. However, an additional layer of context has been applied with the use of a specified domain (Domain 1). DDS entities (data readers/data writers) are restricted to communicate to other entities within that same domain. This provides a useful method of separating traffic when needed by an application. Processes C and D are configured the same way, but operate in Domain 2 and use InfoRepo_2. The challenge comes when you have an application process that needs to use multiple domains and have separate discovery services. This is Process E in our example. It contains two subscribers, one subscribing to publications from InfoRepo_1 and the other subscribing to publications in InfoRepo_2. What allows this configuration to work can be found in the *configE.ini* file.

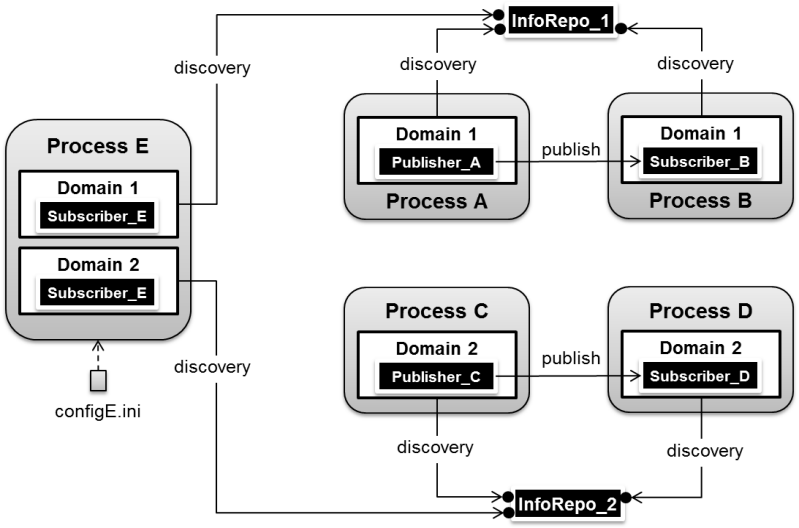


Figure 7-2 Multiple DCPSInfoRepo Configuration

We will now look at the configuration file (referred to as *configE.ini*) to demonstrate how Process E can communicate to both domains and separate DCPSInfoRepo services. For this example we will only show the discovery aspects of the configuration and not show transport content.

configE.ini

```
[domain/1]
DiscoveryConfig=DiscoveryConfig1

[repository/DiscoveryConfig1]
RepositoryIor=host1.mydomain.com:12345

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[repository/DiscoveryConfig2]
RepositoryIor=host2.mydomain.com:12345
```

When Process E in Figure 7-2 reads in the above configuration is finds the occurrence of multiple domain sections. As described in Section each domain has an instance integer and a property of DiscoveryConfig defined.

For the first domain ([domain/1]) The DiscoveryConfig property is supplied with the user-defined name of DiscoveryConfig1 value. This property causes the OpenDDS implementation to find a section title of either repository or rtps_discovery and an instance name of DiscoveryConfig1. In our example, a [repository/DiscoveryConfig1] section title is found and this becomes the discovery configuration for domain instance [domain/1] (integer value 1). The section found now tells us that the address of the DCPSInfoRepo that this domain should use can be found by using the RepositoryIor

property value. In particular it is *host1.mydomain.com* and port *12345*. The values of the RepositoryIor can be a full CORBA IOR or a simple *host:port* string.

A second domain section title [domain/2] is found in this configuration file along with it's corresponding repository section [repository/DiscoveryConfig2] that represents the configuration for the second domain of interest and the InfoRepo_2 repository. There may be any number of repository or domain sections within a single configuration file.

Note *Domains not explicitly configured are automatically associated with the default discovery configuration.*

Note *Individual DCPSInfoRepos can be associated with multiple domains, however domains cannot be shared between multiple DCPSInfoRepos.*

Here are the valid properties for a [repository] section.

Table 7-4 Multiple repository configuration sections

Subsection	Key	Value
[repository/<NAME>]	RepositoryIor	Repository IOR.
	RepositoryKey	Unique key value for the repository. (Deprecated. Provided for backward compatibility)

7.3.3 Configuring For DDS-RTPS Discovery

The OMG DDS-RTPS specification gives the following simple description that forms the basis for the discovery approach used by OpenDDS and the two different protocols used to accomplish the discovery operations. The excerpt from the OMG DDS-RTPS specification Section 8.5.1 is as follows:

> "The RTPS specification splits up the discovery protocol into two independent protocols:
>
> 1. Participant Discovery Protocol
> 2. Endpoint Discovery Protocol
>
> A Participant Discovery Protocol (PDP) specifies how Participants discover each other in the network. Once two Participants have discovered each other, they exchange information on the Endpoints they contain using an Endpoint Discovery Protocol (EDP). Apart from this causality relationship, both protocols can be considered independent."

The configuration options discussed in this section allow a user to specify property values to change the behavior of the Simple Participant Discovery Protocol (SPDP) and/or the Simple Endpoint Discovery Protocol (SEDP) default settings.

Discovery configuration for DDS-RTPS can be accomplished with simple basic discovery or for multiple domains such as that described in Section .

A simple configuration is achieved by specifying a property in the `[common]` section of our example configuration file.

configE.ini (for RTPS)

```
[common]
DCPSDefaultDiscovery=DEFAULT_RTPS
```

All default values for DDS-RTPS discovery are adopted in this form. A variant of this same basic configuration is to specify a section to hold more specific parameters of RTPS discovery. The following example uses the `[common]` section to point to an instance of an `[rtps_discovery]` section followed by an instance name of `TheRTPSConfig` which is supplied by the user.

```
[common]
DCPSDefaultDiscovery=TheRTPSConfig

[rtps_discovery/TheRTPSConfig]
ResendPeriod=5
```

The instance `[rtps_discovery/TheRTPSConfig]` is now the location where properties that vary the default DDS-RTPS settings get specified. In our example the `ResendPeriod=5` entry sets the number of seconds between periodic announcements of available data readers / data writers and to detect the presence of other data readers / data writers on the network. This would override the default of 30 seconds.

If your OpenDDS deployment uses multiple domains, the following configuration approach combines the use of the `[domain]` section title with `[rtps_discovery]` to allow a user to specify particular settings by domain. It might look like this:

configE.ini

```
[common]
DCPSDebugLevel=0

[domain/1]
DiscoveryConfig=DiscoveryConfig1

[rtps_discovery/DiscoveryConfig1]
ResendPeriod=5

[domain/2]
DiscoveryConfig=DiscoveryConfig2

[rtps_discovery/DiscoveryConfig2]
ResendPeriod=5
SedpMulticast=0
```

Some important implementation notes regarding DDS-RTPS discovery in OpenDDS are as follows:

1) Domain IDs should be between 0 and 231 (inclusive) due to the way UDP ports are assigned to domain IDs. In each OpenDDS process, up to 120 domain participants are supported in each domain.

2) OpenDDS's native multicast transport does not work with RTPS Discovery due to the way GUIDs are assigned (a warning will be issued if this is attempted).

The OMG DDS-RTPS specification details several properties that can be adjusted from their defaults that influences the behavior of DDS-RTPS discovery. Find them in Table 7-5.

Table 7-5 RTPS Discovery Configuration Options

Option	Description	Default
`ResendPeriod=sec`	The number of seconds that a process waits between the announcement of participants (see section 8.5.3 in the OMG DDS-RTPS specification for details).	`30`
`PB=port`	Port Base number. This number sets the starting point for deriving port numbers used for Simple Endpoint Discovery Protocol (SEDP). This property is used in conjunction with `DG`, `PG`, `D0` (or `DX`), and `D1` to construct the necessary Endpoints for RTPS discovery communication. (see section 9.6.1.1 in the OMG DDS-RTPS specification in how these Endpoints are constructed)	`7400`
`DG=n`	An integer value representing the Domain Gain. This is a multiplier that assists in formulating Multicast or Unicast ports for RTPS.	`250`
`PG=n`	An integer that assists in configuring SPDP Unicast ports and serves as an offset multiplier as participants are assigned addresses using the formula: `PB + DG * domainId + d1 + PG * participantId` (see section 9.6.1.1 in the OMG DDS-RTPS specification in how these Endpoints are constructed)	2
`D0=n`	An integer value that assists in providing an offset for calculating an assignable port in SPDP Multicast configurations. The formula used is: PB + DG * domainId + d0 (see section 9.6.1.1 in the OMG DDS-RTPS specification in how these Endpoints are constructed)	`0`

Option	Description	Default
`D1=n`	An integer value that assists in providing an offset for calculating an assignable port in SPDP Unicast configurations. The formula used is: `PB + DG * domainId + d1 + PG * participantId` (see section 9.6.1.1 in the OMG DDS-RTPS specification in how these Endpoints are constructed)	`10`
`SedpMulticast= [0|1]`	A boolean value (0 or 1) that determines whether Multicast is used for the SEDP traffic. When set to 1, Multicast is used. When set to zero (0) Unicast for SEDP is used.	`1`
`DX=n`	An integer value that assists in providing an offset for calculating an assignable port in SEDP Multicast configurations. The formula used is: `PB + DG * domainId + dx` This is only valid when `SedpMulticast=1`.This is an OpenDDS extension and not part of the OMG DDS-RTPS specification.	`2`
`SpdpSendAddrs= [host:port],[host:port]...`	A list (comma or whitespace separated) of host:port pairs used as destinations for SPDP content. This can be a combination of Unicast and Multicast addresses.	
InteropMulticastOverride= group_address	A network address specifying the multicast group to be used for SPDP discovery. This overrides the interoperability group of the specification. It ca be used, for example, to specify use of a routed group address to provide a larger discovery scope.	`239.255.0.1`
`TTL=n`	The value of the Time-To-Live (TTL) field of multicast datagrams sent as part of discovery. This value specifies the number of hops the datagram will traverse before being discarded by the network. The default value of 1 means that all data is restricted to the local network subnet.	`1`

Note *If the environment variable* `OPENDDS_RTPS_DEFAULT_D0` *is set, its value is used as the* `D0` *default value.*

7.4 Transport Configuration

Beginning with OpenDDS 3.0, a new transport configuration design has been implemented. The basic goals of this design were to:

- Allow simple deployments to ignore transport configuration and deploy using intelligent defaults (with no transport code required in the publisher or subscriber).
- Enable flexible deployment of applications using only configuration files and command line options.
- Allow deployments that mix transports within individual data writers and writers. Publishers and subscribers negotiate the appropriate transport implementation to use based on the details of the transport configuration, QoS settings, and network reachability.
- Support a broader range of application deployments in complex networks.
- Support optimized transport development (such as collocated and shared memory transports - note that these are not currently implemented).
- Integrate support for the `RELIABILITY` QoS policy with the underlying transport.
- Whenever possible, avoid dependence on the ACE Service Configurator and its configuration files.

Unfortunately, implementing these new capabilities involved breaking of backward compatibility with OpenDDS transport configuration code and files from previous releases. See *`$DDS_ROOT/docs/OpenDDS_3.0_Transition.txt`* for information on how to convert your existing application to use the new transport configuration design.

7.4.1 Overview

7.4.1.1 Transport Concepts

This section provides an overview of the concepts involved in transport configuration and how they interact.

Each data reader and writer uses a *Transport Configuration* consisting of an ordered set of *Transport Instances*. Each Transport Instance specifies a Transport Implementation (i.e. tcp, udp, multicast, shmem, or rtps_udp) and can customize the configuration parameters defined by that transport. Transport Configurations and Transport Instances are managed by the *Transport Registry* and can be created via configuration files or through programming APIs.

Transport Configurations can be specified for Domain Participants, Publishers, Subscribers, Data Writers, and Data Readers. When a Data Reader or Writer is enabled, it uses the most specific configuration it can locate, either directly bound to it or accessible through its parent entity. For example, if a Data Writer specifies a Transport Configuration, it always uses it. If the Data Writer does not specify a configuration, it tries to use that of its

Publisher or Domain Participant in that order. If none of these entities have a transport configuration specified, the *Global Transport Configuration* is obtained from the Transport Registry. The Global Transport Configuration can be specified by the user via either configuration file, command line option, or a member function call on the Transport Registry. If not defined by the user, a default transport configuration is used which contains all available transport implementations with their default configuration parameters. If you don't specifically load or link in any other transport implementations, OpenDDS uses the tcp transport for all communication.

7.4.1.2 How OpenDDS Selects a Transport

Currently, the behavior for OpenDDS is that Data Writers actively connect to Data Readers, which are passively awaiting those connections. Data Readers "listen" for connections on each of the Transport Instances that are defined in their Transport Configuration. Data Writers use their Transport Instances to "connect" to those of the Data Readers. Because the logical connections discussed here don't correspond to the physical connections of the transport, OpenDDS often refers to them as *Data Links*.

When a Data Writer tries to connect to a Data Reader, it first attempts to see if there is an existing data link that it can use to communicate with that Data Reader. The Data Writer iterates (in definition order) through each of its Transport Instances and looks for an existing data link to the Transport Instances that the reader defined. If an existing data link is found it is used for all subsequent communication between the Data Writer and Reader.

If no existing data link is found, the Data Writer attempts to connect using the different Transport Instances in the order they are defined in its Transport Configuration. Any Transport Instances not "matched" by the other side are skipped. For example, if the writer specifies udp and tcp transport instances and the reader only specifies tcp, the udp transport instance is ignored. Matching algorithms may also be affected by QoS parameters, configuration of the instances, and other specifics of the transport implementation. The first pair of Transport Instances that successfully "connect" results in a data link that is used for all subsequent data sample publication.

7.4.2 Configuration File Examples

The following examples explain the basic features of transport configuration via files and describe some common use cases. These are followed by full reference documentation for these features.

7.4.2.1 Single Transport Configuration

The simplest way to provide a transport configuration for your application is to use the OpenDDS configuration file. Here is a sample configuration file that might be used by an

application running on a computer with two network interfaces that only wants to communicate using one of them:

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=mytcp

[transport/mytcp]
transport_type=tcp
local_address=myhost
```

This file does the following (starting from the bottom up):

1) Defines a transport instance named `mytcp` with a transport type of tcp and the local address specified as `myhost`, which is the host name corresponding to the network interface we want to use.

2) Defines a transport configuration named `myconfig` that uses the transport instance `mytcp` as its only transport.

3) Makes the transport configuration named `myconfig` the global transport configuration for all entities in this process.

A process using this configuration file utilizes our customized transport configuration for all Data Readers and Writers created by it (unless we specifically bind another configuration in the code as described in 7.4.2.3).

7.4.2.2 Using Mixed Transports

This example configures an application to primarily use multicast and to “fall back” to tcp when it is unable to use multicast. Here is the configuration file:

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=mymulticast,mytcp

[transport/mymulticast]
transport_type=multicast

[transport/mytcp]
transport_type=tcp
```

The transport configuration named `myconfig` now includes two transport instances, `mymulticast` and `mytcp`. Neither of these transport instances specify any parameters besides transport_type, so they use the default configuration of these transport implementations. Users are free to use any of the transport-specific configuration parameters that are listed in the following reference sections.

Assuming that all participating processes use this configuration file, the application attempts to use multicast to initiate communication between data writers and readers. If the initial multicast communication fails for any reason (possibly because an intervening router is not passing multicast traffic) tcp is used to initiate the connection.

7.4.2.3 Using Multiple Configurations

For many applications, one configuration is not equally applicable to all communication within a given process. These applications must create multiple Transport Configurations and then assign them to the different entities of the process.

For this example consider an application hosted on a computer with two network interfaces that requires communication of some data over one interface and the remainder over the other interface. Here is our configuration file:

```
[common]
DCPSGlobalTransportConfig=config_a

[config/config_a]
transports=tcp_a

[config/config_b]
transports=tcp_b

[transport/tcp_a]
transport_type=tcp
local_address=hosta

[transport/tcp_b]
transport_type=tcp
local_address=hostb
```

Assuming `hosta` and `hostb` are the host names assigned to the two network interfaces, we now have separate configurations that can use tcp on the respective networks. The above file sets the "`A`" side configuration as the default, meaning we must manually bind any entities we want to use the other side to the "`B`" side configuration.

OpenDDS provides two mechanisms to assign configurations to entities:

- Via source code by attaching a configuration to an entity (reader, writer, publisher, subscriber, or domain participant)
- Via configuration file by associating a configuration with a domain

Here is the source code mechanism (using a domain participant):

```
DDS::DomainParticipant_var dp =
        dpf->create_participant(MY_DOMAIN,
                                PARTICIPANT_QOS_DEFAULT,
                                DDS::DomainParticipantListener::_nil(),
                                OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

```
OpenDDS::DCPS::TransportRegistry::instance()->bind_config("config_b", dp);
```

Any Data Writers or Readers owned by this Domain Participant should now use the "B" side configuration.

Note *When directly binding a configuration to a data writer or reader, the* `bind_config` *call must occur before the reader or writer is enabled. This is not an issue when binding configurations to Domain Participants, Publishers, or Subscribers. See Section 3.2.16 for details on how to create entities that are not enabled.*

7.4.3 Transport Registry Example

OpenDDS allows developers to also define transport configurations and instances via C++ APIs. The `OpenDDS::DCPS::TransportRegistry` class is used to construct `OpenDDS::DCPS::TransportConfig` and `OpenDDS::DCPS::TransportInst` objects. The `TransportConfig` and `TransportInst` classes contain public data member corresponding to the options defined below. This section contains the code equivalent of the simple transport configuration file described in . First, we need to include the correct header files:

```
#include <dds/DCPS/transport/framework/TransportRegistry.h>
#include <dds/DCPS/transport/framework/TransportConfig.h>
#include <dds/DCPS/transport/framework/TransportInst.h>
#include <dds/DCPS/transport/tcp/TcpInst.h>

using namespace OpenDDS::DCPS;
```

Next we create the transport configuration, create the transport instance, configure the transport instance, and then add the instance to the configuration's collection of instances:

```
TransportConfig_rch cfg = TheTransportRegistry->create_config("myconfig");
TransportInst_rch inst = TheTransportRegistry->create_inst("mytcp", // name
                                                           "tcp");  // type

// Must cast to TcpInst to get access to transport-specific options
TcpInst_rch tcp_inst = dynamic_rchandle_cast<TcpInst>(inst);
tcp_inst->local_address_str_ = "myhost";

// Add the inst to the config
cfg->instances_.push_back(inst);
```

Lastly, we can make our newly defined transport configuration the global transport configuration:

```
TheTransportRegistry->global_config("myconfig");
```

This code should be executed before any Data Readers or Writers are enabled.

See the header files included above for the full list of public data members and member functions that can be used. See the option descriptions in the following sections for a full understanding of the semantics of these settings.

Stepping back and comparing this code to the original configuration file from , the configuration file is much simpler than the corresponding C++ code and has the added advantage of being modifiable at run-time. It is easy to see why we recommend that almost all applications should use the configuration file mechanism for transport configuration.

7.4.4 Transport Configuration Options

Transport Configurations are specified in the OpenDDS configuration file via sections with the format of `[config/<name>]`, where `<name>` is a unique name for that configuration within that process. The following table summarizes the options when specifying a transport configuration:

Table 7-6 Transport Configuration Options

Option	Description	Default
`Transports=`*`inst1[,inst2][,...]`*	The ordered list of transport instance names that this configuration will utilize. This field is required for every transport configuration.	none
`swap_bytes=`*`[0|1]`*	A value of 0 causes DDS to serialize data in the source machine's native endianness; a value of 1 causes DDS to serialize data in the opposite endianness. The receiving side will adjust the data for its endianness so there is no need to match this option between machines. The purpose of this option is to allow the developer to decide which side will make the endian adjustment, if necessary.	`0`
`passive_connect_duration=`*`msec`*	Timeout (milliseconds) for initial passive connection establishment. By default, this option waits for ten seconds. A value of zero would wait indefinitely (not recommended).	`10000`

The `passive_connect_duration` option is typically set to a non-zero, positive integer. Without a suitable connection timeout, the subscriber endpoint can potentially enter a state of deadlock while waiting for the remote side to initiate a connection. Because there can be multiple transport instances on both the publisher and subscriber side, this option needs to be set to a high enough value to allow the publisher to iterate through the combinations until it succeeds.

In addition to the user-defined configurations, OpenDDS can implicitly define two transport configurations. The first is the default configuration and includes all transport

implementations that are linked into the process. If none are found, then only tcp is used. Each of these transport instances uses the default configuration for that transport implementation. This is the global transport configuration used when the user does not define one.

The second implicit transport configuration is defined whenever an OpenDDS configuration file is used. It is given the same name as the file being read and includes all the transport instances defined in that file, in the alphabetical order of their names. The user can most easily utilize this configuration by specifying the `DCPSGlobalTransportConfiguration=$file` option in the same file. The $file value always binds to the implicit file configuration of the current file.

7.4.5 Transport Instance Options

Transport Instances are specified in the OpenDDS configuration file via sections with the format of `[transport/<name>]`, where `<name>` is a unique name for that instance within that process. Each Transport Instance must specify the `transport_type` option with a valid transport implementation type. The following sections list the other options that can be specified, starting with those options common to all transport types and following with those specific to each transport type.

When using dynamic libraries, the OpenDDS transport libraries are dynamically loaded whenever an instance of that type is defined in a configuration file. When using custom transport implementations or static linking, the application developer is responsible for ensuring that the transport implementation code is linked with their executables.

7.4.5.1 Configuration Options Common to All Transports

The following table summarizes the transport configuration options that are common to all transports:

Table 7-7 Common Transport Configuration Options

Option	Description	Default
`transport_type=`*`transport`*	Type of the transport; the list of available transports can be extended programmatically via the transport framework. tcp, udp, multicast, shmem, and rtps_udp are included with OpenDDS.	none
`queue_messages_per_pool=`*`n`*	When backpressure is detected, messages to be sent are queued. When the message queue must grow, it grows by this number.	`10`

Option	Description	Default
`queue_initial_pools=`*n*	The initial number of pools for the backpressure queue. The default settings of the two backpressure queue values preallocate space for 50 messages (5 pools of 10 messages).	`5`
`max_packet_size=`*n*	The maximum size of a transport packet, including its transport header, sample header, and sample data.	`2147481599`
`max_samples_per_packet=`*n*	Maximum number of samples in a transport packet.	`10`
`optimum_packet_size=`*n*	Transport packets greater than this size will be sent over the wire even if there are still queued samples to be sent. This value may impact performance depending on your network configuration and application nature.	`4096`
`thread_per_connection=` *[0|1]*	Enable or disable the thread per connection send strategy. By default, this option is disabled.	`0`
`datalink_release_delay=` *sec*	The datalink_release_delay is the delay (in seconds) for datalink release after no associations. Increasing this value may reduce the overhead of re-establishment when reader/writer associations are added and removed frequently.	`10`

Enabling the `thread_per_connection` option will increase performance when writing to multiple data readers on different process as long as the overhead of thread context switching does not outweigh the benefits of parallel writes. This balance of network performance to context switching overhead is best determined by experimenting. If a machine has multiple network cards, it may improve performance by creating a transport for each network card.

7.4.5.2 TCP/IP Transport Configuration Options

There are a number of configurable options for the tcp transport. A properly configured transport provides added resilience to underlying stack disturbances. Almost all of the options available to customize the connection and reconnection strategies have reasonable defaults, but ultimately these values should to be chosen based upon a careful study of the quality of the network and the desired QoS in the specific DDS application and target environment.

The local_address option is used by the peer to establish a connection. By default, the TCP transport selects an ephemeral port number on the NIC with the FQDN (fully qualified domain name) resolved. Therefore, you may wish to explicitly set the address if you have multiple NICs or if you wish to specify the port number. When you configure inter-host

communication, the local_address can not be localhost and should be configured with an externally visible address (i.e. 192.168.0.2), or you can leave it unspecified in which case the FQDN and an ephemeral port will be used.

FQDN resolution is dependent upon system configuration. In the absence of a FQDN (e.g. *example.ociweb.com*), OpenDDS will use any discovered short names (e.g. example). If that fails, it will use the name resolved from the loopback address (e.g. localhost).

Note *OpenDDS IPv6 support requires that the underlying ACE/TAO components be built with IPv6 support enabled. The local_address needs to be an IPv6 decimal address or a FQDN with port number. The FQDN must be resolvable to an IPv6 address.*

The `tcp` transport exists as an independent library and needs to be linked in order to use it. When using a dynamically-linked build, OpenDDS automatically loads the transport library whenever it is referenced in a configuration file or as the default transport when no other transports are specified.

When the `tcp` library is built statically, your application must link directly against the library. To do this, your application must first include the proper header for service initialization: <*dds/DCPS/transport/tcp/Tcp.h*>.

You can also configure the publisher and subscriber transport implementations programatically, as described in . Configuring subscribers and publishers should be identical, but different addresses/ports should be assigned to each Transport Instance.

The following table summarizes the transport configuration options that are unique to the `tcp` transport:

Table 7-8 TCP/IP Configuration Options

Option	Description	Default
`conn_retry_attempts=`*n*	Number of reconnect attempts before giving up and calling the `on_publication_lost()` and `on_subscription_lost()` callbacks.	`3`
`conn_retry_initial_delay=`*msec*	Initial delay (milliseconds) for reconnect attempt. As soon as a lost connection is detected, a reconnect is attempted. If this reconnect fails, a second attempt is made after this specified delay.	`500`

Option	Description	Default
`conn_retry_backoff_multiplier=`*`n`*	The backoff multiplier for reconnection tries. After the initial delay described above, subsequent delays are determined by the product of this multiplier and the previous delay. For example, with a conn_retry_initial_delay of 500 and a conn_retry_backoff_multiplier of 1.5, the second reconnect attempt will be 0.5 seconds after the first retry connect fails; the third attempt will be 0.75 seconds after the second retry connect fails; the fourth attempt will be 1.125 seconds after the third retry connect fails.	`2.0`
`enable_nagle_algorithm=`*`[0|1]`*	Enable or disable the Nagle's algorithm. By default, it is disabled. Enabling the Nagle's algorithm may increase throughput at the expense of increased latency.	`0`
`local_address=`*`host:port`*	Hostname and port of the connection acceptor. The default value is the FQDN and port 0, which means the OS will choose the port. If only the host is specified and the port number is omitted, the ':' is still required on the host specifier.	`fqdn:0`
`max_output_pause_period=`*`msec`*	Maximum period (milliseconds) of not being able to send queued messages. If there are samples queued and no output for longer than this period then the connection will be closed and `on_*_lost()` callbacks will be called. The default value of zero means that this check is not made.	`0`
`passive_reconnect_duration=`*`msec`*	The time period (milliseconds) for the passive connection side to wait for the connection to be reconnected. If not reconnected within this period then the `on_*_lost()` callbacks will be called.	`2000`

TCP/IP Reconnection Options

When a TCP/IP connection gets closed OpenDDS attempts to reconnect. The reconnection process is (a successful reconnect ends this sequence):

- Upon detecting a lost connection immediately attempt reconnect.
- If that fails, then wait `conn_retry_initial_delay` milliseconds and attempt reconnect.
- While we have not tried more than `conn_retry_attempts`, wait (previous wait time * `conn_retry_backoff_multiplier`) milliseconds and attempt to reconnect.

7.4.5.3 UDP/IP Transport Configuration Options

The `udp` transport is a bare bones transport that supports best-effort delivery only. Like `tcp`, `local_address`, it supports both IPv4 and IPv6 addresses.

`udp` exists as an independent library and therefore needs to be linked and configured like other transport libraries. When using a dynamic library build, OpenDDS automatically loads the library when it is referenced in a configuration file. When the `udp` library is built statically, your application must link directly against the library. Additionally, your application must also include the proper header for service initialization: *<dds/DCPS/transport/udp/Udp.h>*.

The following table summarizes the transport configuration options that are unique to the `udp` transport:

Table 7-9 UDP/IP Configuration Options

Option	Description	Default
`local_address=`*`host:port`*	Hostname and port of the listening socket. Defaults to a value picked by the underlying OS.	`fqdn:0`
`send_buffer_size=`*`n`*	Total send buffer size in bytes for UDP payload.	`Platform value of ACE_DEFAULT_MAX_SOCKET_BUFSIZ`
`rcv_buffer_size=`*`n`*	Total receive buffer size in bytes for UDP payload.	`Platform value of ACE_DEFAULT_MAX_SOCKET_BUFSIZ`

7.4.5.4 IP Multicast Transport Configuration Options

The `multicast` transport provides unified support for best-effort and reliable delivery based on a transport configuration parameter.

Best-effort delivery imposes the least amount of overhead as data is exchanged between peers, however it does not provide any guarantee of delivery. Data may be lost due to unresponsive or unreachable peers or received in duplicate.

Reliable delivery provides for guaranteed delivery of data to associated peers with no duplication at the cost of additional processing and bandwidth. Reliable delivery is achieved through two primary mechanisms: 2-way peer handshaking and negative acknowledgment of missing data. Each of these mechanisms are bounded to ensure deterministic behavior and is configurable to ensure the broadest applicability possible for user environments.

`multicast` supports a number of configuration options:

The `default_to_ipv6` and `port_offset` options affect how default multicast group addresses are selected. If `default_to_ipv6` is set to “1” (enabled), then the default IPv6

address will be used (`[FF01::80]`). The `port_offset` option determines the default port used when the group address is not set and defaults to `49152`.

The `group_address` option may be used to manually define a multicast group to join to exchange data. Both IPv4 and IPv6 addresses are supported. As with `tcp`, OpenDDS IPv6 support requires that the underlying ACE/TAO components be built with IPv6 support enabled.

On hosts with multiple network interfaces, it may be necessary to specify that the multicast group should be joined on a specific interface. The option `local_address` can be set to the IP address of the local interface that will receive multicast traffic.

If reliable delivery is desired, the `reliable` option may be specified (the default). The remainder of configuration options affect the reliability mechanisms used by the `multicast` transport:

The `syn_backoff`, `syn_interval`, and `syn_timeout` configuration options affect the handshaking mechanism. `syn_backoff` is the exponential base used when calculating the backoff delay between retries. The `syn_interval` option defines the minimum number of milliseconds to wait before retrying a handshake. The `syn_timeout` defines the maximum number of milliseconds to wait before giving up on the handshake.

Given the values of `syn_backoff` and `syn_interval`, it is possible to calculate the delays between handshake attempts (bounded by `syn_timeout`):

```
delay = syn_interval * syn_backoff ^ number_of_retries
```

For example, if the default configuration options are assumed, the delays between handshake attempts would be: 0, 250, 1000, 2000, 4000, and 8000 milliseconds respectively.

The `nak_depth`, `nak_interval`, and `nak_timeout` configuration options affect the Negative Acknowledgment mechanism. `nak_depth` determines the maximum number of datagrams retained by the transport to service incoming repair requests. The `nak_interval` configuration option defines the minimum number of milliseconds to wait between repair requests. This interval is randomized to prevent potential collisions between similarly associated peers. The *maximum* delay between repair requests is bounded to double the minimum value.

The `nak_timeout` configuration option defines the maximum amount of time to wait on a repair request before giving up.

The `nak_delay_intervals` configuration option defines the number of intervals between naks after the initial nak.

The `nak_max` configuration option limits the maximum number of times a missing sample will be nak'ed. Use this option so that naks will be not be sent repeatedly for unrecoverable packets before `nak_timeout`.

Currently, there are a couple of requirements above and beyond those already mandated by the ETF when using this transport:

- *At most*, one DDS domain may be used per multicast group;
- A given participant may only have a single `multicast` transport attached per multicast group; if you wish to send and receive samples on the same multicast group in the same process, independent participants must be used.

`multicast` exists as an independent library and therefore needs to be linked and configured like other transport libraries. When using a dynamic library build, OpenDDS automatically loads the library when it is referenced in a configuration file. When the `multicast` library is built statically, your application must link directly against the library. Additionally, your application must also include the proper header for service initialization: *<dds/DCPS/transport/multicast/Multicast.h>*.

The following table summarizes the transport configuration options that are unique to the `multicast` transport:

Table 7-10 IP Multicast Configuration Options

Option	Description	Default
`default_to_ipv6=`*`[0|1]`*	Enables IPv6 default group address selection. By default, this option is disabled.	`0`
`group_address=`*`host:port`*	The multicast group to join to send/receive data.	`224.0.0.128:<port>, [FF01::80]:<port>`
`local_address=`*`address`*	If non-empty, address of a local network interface which is used to join the multicast group.	
`nak_delay_intervals=`*`n`*	The number of intervals between naks after the initial nak.	`4`
`nak_depth=`*`n`*	The number of datagrams to retain in order to service repair requests (reliable only).	`32`
`nak_interval=`*`msec`*	The minimum number of milliseconds to wait between repair requests (reliable only).	`500`
`nak_max=`*`n`*	The maximum number of times a missing sample will be nak'ed.	`3`
`nak_timeout=`*`msec`*	The maximum number of milliseconds to wait before giving up on a repair response (reliable only). The default is 30 seconds.	`30000`

Option	Description	Default
`port_offset=`*n*	Used to set the port number when not specifying a group address. When a group address is specified, the port number within it is used. If no group address is specified, the port offset is used as a port number. This value should not be set less than 49152.	`49152`
`rcv_buffer_size=`*n*	The size of the socket receive buffer in bytes. A value of zero indicates that the system default value is used.	`0`
`reliable=`*[0|1]*	Enables reliable communication.	`1`
`syn_backoff=`*n*	The exponential base used during handshake retries; smaller values yield shorter delays between attempts.	`2.0`
`syn_interval=`*msec*	The minimum number of milliseconds to wait between handshake attempts during association.	`250`
`syn_timeout=`*msec*	The maximum number of milliseconds to wait before giving up on a handshake response during association. The default is 30 seconds.	`30000`
`ttl=`*n*	The value of the time-to-live (ttl) field of any datagrams sent. The default value of one means that all data is restricted to the local network.	`1`

7.4.5.5 RTPS_UDP Transport Configuration Options

The OpenDDS implementation of the OMG DDS-RTPS (`formal/2010-11-01`) specification includes the transport protocols necessary to fulfill the specification requirements and those needed to be interoperable with other DDS implementations. The `rtps_udp` transport is one of the pluggable transports available to a developer and is necessary for interoperable communication between implementations. This section will discuss the options available to the developer for configuring OpenDDS to use this transport.

To provide an RTPS variant of the single configuration example from Section , the configuration file below simply modifies the `transport_type` property to the value `rtps_udp`. All other items remain the same.

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=myrtps

[transport/myrtps]
transport_type=rtps_udp
local_address=myhost
```

To extend our examples to a mixed transport configuration as shown in Section , below shows the use of an `rtps_udp` transport mixed with a `tcp` transport. The interesting pattern that this allows for is a deployed OpenDDS application that can be, for example, communicating using `tcp` with other OpenDDS participants while communicating in an interoperability configuration with a non-OpenDDS participant using `rtps_udp`.

```
[common]
DCPSGlobalTransportConfig=myconfig

[config/myconfig]
transports=mytcp,myrtps

[transport/myrtps]
transport_type=rtps_udp

[transport/mytcp]
transport_type=tcp
```

Some implementation notes related to using the `rtps_udp` transport protocol are as follows:

1) `WRITER_DATA_LIFECYCLE` (8.7.2.2.7) notes that the same Data sub-message should dispose and unregister an instance. OpenDDS may use two Data sub-messages.

2) RTPS transport instances can not be shared by different Domain Participants.

3) Transport auto-selection (negotiation) is partially supported with RTPS such that the `rtps_udp` transport goes through a handshaking phase only in reliable mode.

Table 7-11 RTPS_UDP Configuration Options

Option	Description	Default
`use_multicast=`*`[0|1]`*	The `rtps_udp` transport can use Unicast or Multicast. When set to 0 (false) the transport uses Unicast, otherwise a value of 1 (true) will use Multicast.	`1`
`multicast_group_address =`*`network address`*	When the transport is set to multicast, this is the multicast network address that should be used. If no port is specified for the network address, port 7401 will be used.	`239.255.0.2:7401`
`nak_depth=`*`n`*	The number of datagrams to retain in order to service repair requests (reliable only).	`32`
`nak_response_delay=`*`msec`*	Protocol tuning parameter that allows the RTPS Writer to delay the response (expressed in milliseconds) to a request for data from a negative acknowledgment. (see table 8.47 in the OMG DDS-RTPS specification)	`200`

Option	Description	Default
`heartbeat_period=`*`msec`*	Protocol tuning parameter that specifies in milliseconds how often an RTPS Writer announces the availability of data. (see table 8.47 in the OMG DDS-RTPS specification)	`1000`
`heartbeat_response_delay=`*`msec`*	Protocol tuning parameter in milliseconds that allows the RTPS Reader to delay the sending of a positive or negative acknowledgment. This parameter is used to reduce the occurrences of network storms. (see Section 8.4.12.2 of the OMG DDS-RTPS specification (`formal/2010-11-01`))	`500`
`handshake_timeout=`*`msec`*	The maximum number of milliseconds to wait before giving up on a handshake response during association. The default is 30 seconds.	`30000`
`ttl=`*`n`*	The value of the time-to-live (ttl) field of any multicast datagrams sent. This value specifies the number of hops the datagram will traverse before being discarded by the network. The default value of 1 means that all data is restricted to the local network subnet.	`1`

7.4.5.6 Shared-Memory Transport Configuration Options

The following table summarizes the transport configuration options that are unique to the `shmem` transport. This transport type is supported Unix-like platforms with POSIX/XSI shared memory and on Windows platforms. The shared memory transport type can only provide communication between transport instances on the same host. As part of transport negotiation (see 7.4.2.2), if there are multiple transport instances available for communication between hosts, the shared memory transport instances will be skipped so that other types can be used.

Table 7-12 Shared-Memory Transport Configuration Options

Option	Description	Default
`pool_size=`*`bytes`*	The size of the single shared-memory pool allocated.	`16 MB`
`datalink_control_size=`*`bytes`*	The size of the control area allocated for each data link. This allocation comes out of the shared-memory pool defined by pool_size.	`4 KB`

7.5 Logging

By default, the OpenDDS framework will only log when there is a serious error that is not indicated by a return code. An OpenDDS user may increase the amount of logging via controls at the DCPS and Transport layers.

7.5.1 DCPS Layer Logging

Logging in the DCPS layer of OpenDDS is controlled by the `DCPSDebugLevel` configuration option and command-line option. It can also be set programmatically in application code using:

```
OpenDDS::DCPS::set_DCPS_debug_level(level)
```

The *level* defaults to a value of 0 and has values of 0 to 10 as defined below:

- 0 - logs that indicate serious errors that are not indicated by return codes (almost none).
- 1 - logs that should happen once per process or are warnings
- 2 - logs that should happen once per DDS entity
- 4 - logs that are related to administrative interfaces
- 6 - logs that should happen every Nth sample write/read
- 8 - logs that should happen once per sample write/read
- 10 - logs that may happen more than once per sample write/read

7.5.2 Transport Layer Logging

OpenDDS transport layer logging is controlled via the `DCPSTransportDebugLevel` configuration option. For example, to add transport layer logging to any OpenDDS application, add the following option to the command line:

```
-DCPSTransportDebugLevel level;
```

The transport layer logging level can also be programmatically configured by appropriately setting the variable:

```
OpenDDS::DCPS::Transport_debug_level = level;
```

Valid transport logging levels range from 0 to 5 with increasing verbosity of output.

Note *Transport logging level 6 is available to generate system trace logs. Using this level is not recommended as the amount of data generated can be overwhelming and is mostly of interest only to OpenDDS developers. Setting the logging level to 6 requires defining the* `DDS_BLD_DEBUG_LEVEL` *macro to* `6` *and rebuilding OpenDDS.*

CHAPTER 8

opendds_idl Options

8.1 opendds_idl Command Line Options

The OpenDDS IDL compiler is invoked using the `opendds_idl` executable, located in `$DDS_ROOT/bin/`. It parses a single IDL file and generates the serialization and key support code that OpenDDS requires to marshal and demarshal the types described in the IDL file, as well as the type support code for the data readers and writers. For each IDL file processed, such as *`xyz.idl`*, it generates three files: *`xyzTypeSupport.idl`*, *`xyzTypeSupportImpl.h`*, and *`xyzTypeSupportImpl.cpp`*. In the typical usage, `opendds_idl` is passed a number of options and the IDL file name as a parameter. For example,

```
$DDS_ROOT/bin/opendds_idl Foo.idl
```

The following table summarizes the options supported by `opendds_idl`.

Table 8-1 opendds_idl Command Line Options

Option	Description	Default
`-v`	Enables verbose execution	Quiet execution
`-h`	Prints a help (usage) message and exits	N/A
`-V`	Prints version numbers of both TAO and OpenDDS	N/A

Option	Description	Default
`-Wb,export_macro=macro`	Export macro used for generating C++ implementation code.	No export macro used
`-Wb,export_include=file`	Additional header to `#include` in generated code -- this header `#defines` the export macro	No additional include
`-Wb,pch_include=file`	Pre-compiled header file to include in generated C++ files	No pre-compiled header included
`-Dname[=value]`	Define a preprocessor macro	N/A
`-Idir`	Add dir to the preprocessor include path	N/A
`-o outputdir`	Output directory where `opendds_idl` should place the generated files.	The current directory
`-Wb,java`	Enable OpenDDS Java Bindings for generated TypeSupport implementation classes	No Java support
`-Gws`	Generates wireshark dissector configuration files of the form *`<file>_ws.ini`*.	No wireshark dissector configuration generated

The code generation options allow the application developer to use the generated code in a wide variety of environments. Since IDL may contain preprocessing directives (`#include`, `#define`, etc.), the C++ preprocessor is invoked by `opendds_idl`. The `-I` and `-D` options allow customization of the preprocessing step. The `-Wb,export_macro` option(`--export` is accepted as an alias) lets you add an export macro to your class definitions. This is required if the generated code is going to reside in a shared library and the compiler (such as Visual C++ or GCC 4) uses the export macro (`dllexport` on Visual C++ / overriding hidden visibility on GCC 4). The `-Wb,pch_include` option is required if the generated implementation code is to be used in a component that uses precompiled headers.

CHAPTER 9

The DCPS Information Repository

9.1 DCPS Information Repository Options

The table below shows the command line options for the `DCPSInfoRepo` server:

Table 9-1 DCPS Information Repository Options

Option	Description	Default
`-o file`	Write the IOR of the `DCPSInfo` object to the specified file	*`repo.ior`*
`-NOBITS`	Disable the publication of built-in topics	Built-in topics are published
`-a address`	Listening address for built-in topics (when built-in topics are published).	Random port
`-z`	Turn on verbose transport logging	Minimal transport logging.
`-r`	Resurrect from persistent file	1(true)
`-FederationId <id>`	Unique identifier for this repository within any federation. This is supplied as a 32 bit decimal numeric value.	N/A

Option	Description	Default
`-FederateWith <ref>`	Repository federation reference at which to join a federation. This is supplied as a valid CORBA object reference in string form: stringified IOR, file: or corbaloc: reference string.	N/A
`-?`	Display the command line usage and exit	N/A

OpenDDS clients often use the IOR file that `DCPSInfoRepo` outputs to locate the service. The `-o` option allows you to place the IOR file into an application-specific directory or file name. This file can subsequently be used by clients with the *`file://`* IOR prefix.

Applications that do not use built-in topics may want to disable them with `-NOBITS` to reduce the load on the server. If you are publishing the built-in topics, then the `-a` option lets you pick the listen address of the tcp transport that is used for these topics.

Using the `-z` option causes the invocation of many transport-level debug messages. This option is only effective when the DCPS library is built with the `DCPS_TRANS_VERBOSE_DEBUG` environment variable defined.

The `-FederationId` and `-FederateWith` options are used to control the federation of multiple `DCPSInfoRepo` servers into a single logical repository. See 1.2 for descriptions of the federation capabilities and how to use these options.

File persistence is implemented as an ACE Service object and is controlled via service config directives. Currently available configuration options are:

Table 9-2 InfoRepo persistence directives

Options	Description	Defaults
`-file`	Name of the persistent file	*`InforepoPersist`*
`-reset`	Wipe out old persistent data.	`0` (false)

The following directive:

```
static PersistenceUpdater_Static_Service "-file info.pr -reset 1"
```

will persist `DCPSInfoRepo` updates to local file *`info.pr`*. If a file by that name already exists, its contents will be erased. Used with the command-line option `-r`, the `DCPSInfoRepo` can be reincarnated to a prior state. When using persistence, start the `DCPSInfoRepo` process using a TCP fixed port number with the following command line option. This allows existing clients to reconnect to a restarted InfoRepo.

```
-ORBListenEndpoints iiop://:<port>
```

9.2 Repository Federation

Note *Repository federation should be considered an experimental feature.*

Repository Federation allows multiple DCPS Information Repository servers to collaborate with one another into a single federated service. This allows applications obtaining service metadata and events from one repository to obtain them from another if the original repository is no longer available.

While the motivation to create this feature was the ability to provide a measure of fault tolerance to the DDS service metadata, other use cases can benefit from this feature as well. This includes the ability of initially separate systems to become federated and gain the ability to pass data between applications that were not originally reachable. An example of this would include two platforms which have independently established internal DDS services passing data between applications; at some point during operation the systems become reachable to each other and federating repositories allows data to pass between applications on the different platforms.

The current federation capabilities in OpenDDS provide only the ability to statically specify a federation of repositories at startup of applications and repositories. A mechanism to dynamically discover and join a federation is planned for a future OpenDDS release.

OpenDDS automatically detects the loss of a repository by using the `LIVELINESS` Quality of Service policy on a Built-in Topic. When a federation is used, the `LIVELINESS` QoS policy is modified to a non-infinite value. When `LIVELINESS` is lost for a Built-in Topic an application will initiate a failover sequence causing it to associate with a different repository server. Because the federation implementation currently uses a Built-in Topic `ParticipantDataDataReaderListener` entity, applications should not install their own listeners for this topic. Doing so would affect the federation implementation's capability to detect repository failures.

The federation implementation distributes repository data within the federation using a reserved DDS domain. The default domain used for federation is defined by the constant `Federator::DEFAULT_FEDERATIONDOMAIN`.

Currently only static specification of federation topology is available. This means that each DCPS Information Repository, as well as each application using a federated DDS service, needs to include federation configuration as part of its configuration data. This is done by specifying each available repository within the federation to each participating process and assigning each repository to a different key value in the configuration files as described in Section 7.3.2.1.

Each application and repository must include the same set of repositories in its configuration information. Failover sequencing will attempt to reach the next repository in numeric sequence (wrapping from the last to the first) of the repository key values. This sequence is unique to each application configured, and should be different to avoid overloading any individual repository.

Once the topology information has been specified, then repositories will need to be started with two additional command line arguments. These are shown in Table 9-1. One, `-FederationId <value>`, specifies the unique identifier for a repository within the federation. This is a 32 bit numeric value and needs to be unique for all possible federation topologies.

The second command line argument required is `-FederateWith <ref>`. This causes the repository to join a federation at the <ref> object reference after initialization and before accepting connections from applications.

Only repositories which are started with a federation identification number may participate in a federation. The first repository started should not be given a `-FederateWith` command line directive. All others are required to have this directive in order to establish the initial federation. There is a command line tool (`federation`) supplied that can be used to establish federation associations if this is not done at startup. See Section 9.2.1 for a description. It is possible, with the current static-only implementation, that the failure of a repository before a federation topology is entirely established could result in a partially unusable service. Due to this current limitation, it is highly recommended to always establish the federation topology of repositories prior to starting the applications.

9.2.1 Federation Management

A new command line tool has been provided to allow some minimal run-time management of repository federation. This tool allows repositories started without the `-FederateWith` option to be commanded to participate in a federation. Since the operation of the federated repositories and failover sequencing depends on the presence of connected topology, it is recommended that this tool be used before starting applications that will be using the federated set of repositories.

The command is named `repoctl` and is located in the `$DDS_ROOT/bin/` directory. It has a command format syntax of:

```
repoctl <cmd> <arguments>
```

Where each individual command has its own format as shown in Table 9-3. Some options contain endpoint information. This information consists of an optional host specification, separated from a required port specification by a colon. This endpoint information is used to

create a CORBA object reference using the corbaloc: syntax in order to locate the 'Federator' object of the repository server.

Table 9-3 repoctl Repository Management Command

Command	Syntax	Description
`join`	`repoctl join <target> <peer> [ <federation domain> ]`	Calls the `<peer>` to join `<target>` to the federation. `<federation domain>` is passed if present, or the default Federation Domain value is passed.
`leave`	`repoctl leave <target>`	Causes the `<target>` to gracefully leave the federation, removing all managed associations between applications using `<target>` as a repository with applications that are not using `<target>` as a repository.
`shutdown`	`repoctl shutdown <target>`	Causes the `<target>` to shutdown without removing any managed associations. This is the same effect as a repository which has crashed during operation.
`kill`	`repoctl kill <target>`	Kills the `<target>` repository regardless of its federation status.
`help`	`repoctl help`	Prints a usage message and quits.

A join command specifies two repository servers (by endpoint) and asks the second to join the first in a federation:

```
repoctl join 2112 otherhost:1812
```

This generates a CORBA object reference of *`corbaloc::otherhost:1812/Federator`* that the federator connects to and invokes a join operation. The join operation invocation passes the default Federation Domain value (because we did not specify one) and the location of the joining repository which is obtained by resolving the object reference *`corbaloc::localhost:2112/Federator`*.

A full description of the command arguments are shown in Table 9-4.

Table 9-4 Federation Management Command Arguments

Option	Description
`<target>`	This is endpoint information that can be used to locate the `Federator::Manager` CORBA interface of a repository which is used to manage federation behavior. This is used to command leave and shutdown federation operations and to identify the joining repository for the join command.
`<peer>`	This is endpoint information that can be used to locate the `Federator::Manager` CORBA interface of a repository which is used to manage federation behavior. This is used to command join federation operations.

Option	Description
`<federation domain>`	This is the domain specification used by federation participants to distribute service metadata amongst the federated repositories. This only needs to be specified if more than one federation exists among the same set of repositories, which is currently not supported. The default domain is sufficient for single federations.

9.2.2 Federation Example

In order to illustrate the setup and use of a federation, this section walks through a simple example that establishes a federation and a working service that uses it.

This example is based on a two repository federation, with the simple Message publisher and subscriber from 2.1 configured to use the federated repositories.

9.2.2.1 Configuring the Federation Example

There are two configuration files to create for this example one each for the message publisher and subscriber.

The Message Publisher configuration *pub.ini* for this example is as follows:

```
[common]
DCPSDebugLevel=0

[domain/information]
DomainId=42
DomainRepoKey=1

[repository/primary]
RepositoryKey=1
RepositoryIor=corbaloc::localhost:2112/InfoRepo

[repository/secondary]
RepositoryKey=2
RepositoryIor=file://repo.ior
```

Note that the `DCPSInfo` attribute/value pair has been omitted from the `[common]` section. This has been replaced by the `[domain/user]` section as described in 7.5. The user domain is 42, so that domain is configured to use the primary repository for service metadata and events.

The `[repository/primary]` and `[repository/secondary]` sections define the primary and secondary repositories to use within the federation (of two repositories) for this application. The `RepositoryKey` attribute is an internal key value used to uniquely identify the repository (and allow the domain to be associated with it, as in the preceding `[domain/information]` section). The `RepositoryIor` attributes contain string values of resolvable object references to reach the specified repository. The primary repository is referenced at port 2112 of the *localhost* and is expected to be available via the TAO

`IORTable` with an object name of `/InfoRepo`. The secondary repository is expected to provide an IOR value via a file named *`repo.ior`* in the local directory.

The subscriber process is configured with the *`sub.ini`* file as follows:

```
[common]
DCPSDebugLevel=0

[domain/information]
DomainId=42
DomainRepoKey=1

[repository/primary]
RepositoryKey=1
RepositoryIor=file://repo.ior

[repository/secondary]
RepositoryKey=2
RepositoryIor=corbaloc::localhost:2112/InfoRepo
```

Note that this is the same as the *`pub.ini`* file except the subscriber has specified that the repository located at port 2112 of the *`localhost`* is the secondary and the repository located by the *`repo.ior`* file is the primary. This is opposite of the assignment for the publisher. It means that the publisher is started using the repository at port 2112 for metadata and events while the subscriber is started using the repository located by the IOR contained in the file. In each case, if a repository is detected as unavailable the application will attempt to use the other repository if it can be reached.

The repositories do not need any special configuration specifications in order to participate in federation, and so no files are required for them in this example.

9.2.2.2 Running the Federation Example

The example is executed by first starting the repositories and federating them, then starting the application publisher and subscriber processes the same way as was done in the example of Section 2.1.7.

Start the first repository as:

```
$DDS/bin/DCPSInfoRepo -ORBSvcConf tcp.conf -o repo.ior -FederationId 1024
```

The `-o repo.ior` option ensures that the repository IOR will be placed into the file as expected by the configuration files. The `-FederationId 1024` option assigns the value 1024 to this repository as its unique id within the federation. The `-ORBSvcConf tcp.conf` option is the same as in the previous example.

Start the second repository as:

```
$DDS/bin/DCPSInfoRepo -ORBSvcConf tcp.conf \
  -ORBListenEndpoints iiop://localhost:2112 \
```

```
-FederationId 2048 -FederateWith file://repo.ior
```

Note that this is all intended to be on a single command line. The `-ORBSvcConf` *`tcp.conf`* option is the same as in the previous example. The `-ORBListenEndpoints iiop://`*`localhost`*`:2112` option ensures that the repository will be listening on the port that the previous configuration files are expecting. The `-FederationId 2048` option assigns the value 2048 as the repositories unique id within the federation. The `-FederateWith file://repo.ior` option initiates federation with the repository located at the IOR contained within the named file - which was written by the previously started repository.

Once the repositories have been started and federation has been established (this will be done automatically after the second repository has initialized), the application publisher and subscriber processes can be started and should execute as they did for the previous example in Section 2.1.7.

CHAPTER 10

OpenDDS Java Bindings

10.1 Introduction

OpenDDS provides Java JNI bindings. Java applications can make use of the complete OpenDDS middleware just like C++ applications.

See the *$DDS_ROOT/java/INSTALL* file for information on getting started, including the prerequisites and dependencies.

See the *$DDS_ROOT/java/FAQ* file for information on common issues encountered while developing applications with the Java bindings.

10.2 IDL and Code Generation

The OpenDDS Java binding is more than just a library that lives in one or two .jar files. The DDS specification defines the interaction between a DDS application and the DDS middleware. In particular, DDS applications send and receive messages that are strongly-typed and those types are defined by the application developer in IDL.

In order for the application to interact with the middleware in terms of these user-defined types, code must be generated at compile-time based on this IDL. C++, Java, and even some additional IDL code is generated. In most cases, application developers do not need to be concerned with the details of all the generated files. Scripts included with OpenDDS

automate this process so that the end result is a native library (*.so* or *.dll*) and a Java library (*.jar* or just a *classes* directory) that together contain all of the generated code.

Below is a description of the generated files and which tools generate them. In this example, *Foo.idl* contains a single struct `Bar` contained in module `Baz` (IDL modules are similar to C++ namespaces and Java packages). To the right of each file name is the name of the tool that generates it, followed by some notes on its purpose.

Table 10-1 Generated files descriptions

File	Generation Tool
Foo.idl	Developer-written description of the DDS sample type
Foo{C,S}.{h,inl,cpp}	`tao_idl`: C++ representation of the IDL
FooTypeSupport.idl	`opendds_idl`: DDS type-specific interfaces
FooTypeSupport{C,S}.{h,inl,cpp}	`tao_idl`
Baz/BarSeq{Helper,Holder}.java	`idl2jni`
Baz/BarData{Reader,Writer}.java*	`idl2jni`
Baz/BarTypeSupport.java*	`idl2jni` (except TypeSupportImpl, see below)
FooTypeSupportJC.{h,cpp}	`idl2jni`: JNI native method implementations
FooTypeSupportImpl.{h,cpp}	`opendds_idl`: DDS type-specific C++ impl.
Baz/BarTypeSupportImpl.java	`opendds_idl`: DDS type-specific Java impl.
Baz/Bar.java*	`idl2jni`: Java representation of IDL struct
FooJC.{h,cpp}	`idl2jni`: JNI native method implementations

`Foo.idl:`

```
module Baz {
#pragma DCPS_DATA_TYPE "Baz::Bar"
  struct Bar {
    long x;
  };
};
```

10.3 Setting up an OpenDDS Java Project

These instructions assume you have completed the installation steps in the *$DDS_ROOT/java/INSTALL* document, including having the various environment variables defined.

1) Start with an empty directory that will be used for your IDL and the code generated from it. `$DDS_ROOT/java/tests/messenger/messenger_idl/` is set up this way.

2) Create an IDL file describing the data structure you will be using with OpenDDS. See *Messenger.idl* for an example. This file will contain at least one line starting with "`#pragma DCPS_DATA_TYPE`". For the sake of these instructions, we will call the file *Foo.idl*.

3) The C++ generated classes will be packaged in a shared library to be loaded at run-time by the JVM. This requires the packaged classes to be exported for external visibility. ACE provides a utility script for generating the correct export macros. The script usage is shown here:

   Unix:

   ```
   $ACE_ROOT/bin/generate_export_file.pl Foo > Foo_Export.h
   ```

   Windows:

   ```
   %ACE_ROOT%\bin\generate_export_file.pl Foo > Foo_Export.h
   ```

4) Create an MPC file, Foo.mpc, from this template:

```
project: dcps_java {

idlflags      += -Wb,stub_export_include=Foo_Export.h \
                 -Wb,stub_export_macro=Foo_Export
dcps_ts_flags += -Wb,export_macro=Foo_Export
idl2jniflags  += -Wb,stub_export_include=Foo_Export.h \
                 -Wb,stub_export_macro=Foo_Export
dynamicflags  += FOO_BUILD_DLL

specific {
  jarname      = DDS_Foo_types
}

TypeSupport_Files {
  Foo.idl
}
}
```

   You can leave out the specific {...} block if you do not need to create a jar file. In this case you can directly use the Java .class files which will be generated under the classes subdirectory of the current directory.

5) Run MPC to generate platform-specific build files.

   Unix:

   ```
   $ACE_ROOT/bin/mwc.pl -type gnuace
   ```

   Windows:

   ```
   %ACE_ROOT%\bin\mwc.pl -type [CompilerType]
   ```

   CompilerType can be vc71, vc8, vc9, vc10, and nmake

   Make sure this is running ActiveState Perl.

6) Compile the generated C++ and Java code

   Unix:

```
make (GNU make, so this may be "gmake" on Solaris systems)
```

Windows:

Build the generated *.sln* (Solution) file using your preferred method. This can be either the Visual Studio IDE or one of the command-line tools. If you use the IDE, start it from a command prompt using `devenv` or `vcexpress` (Express Edition) so that it inherits the environment variables. Command-line tools for building include `vcbuild` and invoking the IDE (`devenv` or `vcexpress`) with the appropriate arguments.

When this completes successfully you have a native library and a Java *.jar* file. The native library names are as follows:

Unix:

```
libFoo.so
```

Windows:

```
Foo.dll (Release) or Food.dll (Debug)
```

You can change the locations of these libraries (including the *.jar* file) by adding a line such as the following to the *Foo.mpc* file:

```
libout = $(PROJECT_ROOT)/lib
```

where `PROJECT_ROOT` can be any environment variable defined at build-time.

7) You now have all of the Java and C++ code needed to compile and run a Java OpenDDS application. The generated *.jar* file needs to be added to your `classpath`. The generated C++ library needs to be available for loading at run-time:

   Unix:

   Add the directory containing *libFoo.so* to the `LD_LIBRARY_PATH`.

   Windows:

   Add the directory containing *Foo.dll* (or *Food.dll*) to the `PATH`. If you are using the debug version (*Food.dll*) you will need to inform the OpenDDS middleware that it should not look for *Foo.dll*. To do this, add `-Djni.nativeDebug=1` to the Java VM arguments.

   See the publisher and subscriber directories in `$DDS_ROOT/java/tests/messenger/` for examples of publishing and subscribing applications using the OpenDDS Java bindings.

8) If you make subsequent changes to *Foo.idl*, start by re-running MPC (step #5 above). This is needed because certain changes to *Foo.idl* will affect which files are generated and need to be compiled.

10.4 A Simple Message Publisher

This section presents a simple OpenDDS Java publishing process. The complete code for this can be found at *$DDS_ROOT/java/tests/messenger/publisher/TestPublisher.java*. Uninteresting segments such as imports and error handling have been omitted here. The code has been broken down and explained in logical subsections.

10.4.1 Initializing The Participant

DDS applications are boot-strapped by obtaining an initial reference to the Participant Factory. A call to the static method `TheParticipantFactory.WithArgs()` returns a Factory reference. This also transparently initializes the C++ Participant Factory. We can then create Participants for specific domains.

```
public static void main(String[] args) {

    DomainParticipantFactory dpf =
        TheParticipantFactory.WithArgs(new StringSeqHolder(args));
    if (dpf == null) {
      System.err.println ("Domain Participant Factory not found");
      return;
    }
    final int DOMAIN_ID = 42;
    DomainParticipant dp = dpf.create_participant(DOMAIN_ID,
      PARTICIPANT_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
    if (dp == null) {
      System.err.println ("Domain Participant creation failed");
      return;
    }
```

Object creation failure is indicated by a null return. The third argument to `create_participant()` takes a Participant events listener. If one is not available, a null can be passed instead as done in our example.

10.4.2 Registering The Data Type And Creating A Topic

Next we register our data type with the `DomainParticipant` using the `register_type()` operation. We can specify a type name or pass an empty string. Passing an empty string indicates that the middleware should simply use the identifier generated by the IDL compiler for the type.

```
    MessageTypeSupportImpl servant = new MessageTypeSupportImpl();
```

```
if (servant.register_type(dp, "") != RETCODE_OK.value) {
  System.err.println ("register_type failed");
  return;
}
```

Next we create a topic using the type support servant's registered name.

```
Topic top = dp.create_topic("Movie Discussion List",
                            servant.get_type_name(),
                            TOPIC_QOS_DEFAULT.get(), null,
                            DEFAULT_STATUS_MASK.value);
```

Now we have a topic named "*Movie Discussion List*" with the registered data type and default QoS policies.

10.4.3 Creating A Publisher

Next, we create a publisher:

```
Publisher pub = dp.create_publisher(
  PUBLISHER_QOS_DEFAULT.get(),
  null,
  DEFAULT_STATUS_MASK.value);
```

10.4.4 Creating A DataWriter And Registering An Instance

With the publisher, we can now create a DataWriter:

```
DataWriter dw = pub.create_datawriter(
  top, DATAWRITER_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
```

The `DataWriter` is for a specific topic. For our example, we use the default `DataWriter` QoS policies and a null `DataWriterListener`.

Next, we narrow the generic `DataWriter` to the type-specific `DataWriter` and register the instance we wish to publish. In our data definition IDL we had specified the subject_id field as the key, so it needs to be populated with the instance id (99 in our example):

```
MessageDataWriter mdw = MessageDataWriterHelper.narrow(dw);
Message msg = new Message();
msg.subject_id = 99;
int handle = mdw.register(msg);
```

Our example waits for any peers to be initialized and connected. It then publishes a few messages which are distributed to any subscribers of this topic in the same domain.

```
msg.from = "OpenDDS-Java";
msg.subject = "Review";
msg.text = "Worst. Movie. Ever.";
```

```
msg.count = 0;
int ret = mdw.write(msg, handle);
```

10.5 Setting up the Subscriber

Much of the initialization code for a subscriber is identical to the publisher. The subscriber needs to create a participant in the same domain, register an identical data type, and create the same named topic.

```
public static void main(String[] args) {

    DomainParticipantFactory dpf =
        TheParticipantFactory.WithArgs(new StringSeqHolder(args));
    if (dpf == null) {
      System.err.println ("Domain Participant Factory not found");
      return;
    }
    DomainParticipant dp = dpf.create_participant(42,
      PARTICIPANT_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
    if (dp == null) {
      System.err.println("Domain Participant creation failed");
      return;
    }

    MessageTypeSupportImpl servant = new MessageTypeSupportImpl();

    Topic top = dp.create_topic("Movie Discussion List",
                                servant.get_type_name(),
                                TOPIC_QOS_DEFAULT.get(), null,
                                DEFAULT_STATUS_MASK.value);
```

10.5.1 Creating A Subscriber

As with the publisher, we create a subscriber:

```
Subscriber sub = dp.create_subscriber(
  SUBSCRIBER_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
```

10.5.2 Creating A DataReader And Listener

Providing a `DataReaderListener` to the middleware is the simplest way to be notified of the receipt of data and to access the data. We therefore create an instance of a `DataReaderListenerImpl` and pass it as a `DataReader` creation parameter:

```
DataReaderListenerImpl listener = new DataReaderListenerImpl();
 DataReader dr = sub.create_datareader(
   top, DATAREADER_QOS_DEFAULT.get(), listener,
   DEFAULT_STATUS_MASK.value);
```

Any incoming messages will be received by the Listener in the middleware's thread. The application thread is free to perform other tasks at this time.

10.6 The DataReader Listener Implementation

The application defined `DataReaderListenerImpl` needs to implement the specification's `DDS.DataReaderListener` interface. OpenDDS provides an abstract class `DDS._DataReaderListenerLocalBase`. The application's listener class extends this abstract class and implements the abstract methods to add application-specific functionality.

Our example `DataReaderListener` stubs out most of the Listener methods. The only method implemented is the message available callback from the middleware:

```
public class DataReaderListenerImpl extends DDS._DataReaderListenerLocalBase {

    private int num_reads_;

    public synchronized void on_data_available(DDS.DataReader reader) {
        ++num_reads_;
        MessageDataReader mdr = MessageDataReaderHelper.narrow(reader);
        if (mdr == null) {
          System.err.println ("read: narrow failed.");
          return;
        }
```

The Listener callback is passed a reference to a generic `DataReader`. The application narrows it to a type-specific `DataReader`:

```
        MessageHolder mh = new MessageHolder(new Message());
        SampleInfoHolder sih = new SampleInfoHolder(new SampleInfo(0, 0, 0,
            new DDS.Time_t(), 0, 0, 0, 0, 0, 0, 0, false));
        int status  = mdr.take_next_sample(mh, sih);
```

It then creates holder objects for the actual message and associated `SampleInfo` and takes the next sample from the `DataReader`. Once taken, that sample is removed from the `DataReader`'s available sample pool.

```
        if (status == RETCODE_OK.value) {

          System.out.println ("SampleInfo.sample_rank = "+ sih.value.sample_rank);
          System.out.println ("SampleInfo.instance_state = "+
                              sih.value.instance_state);

          if (sih.value.valid_data) {

            System.out.println("Message: subject    = " + mh.value.subject);
            System.out.println("         subject_id = " + mh.value.subject_id);
            System.out.println("         from       = " + mh.value.from);
            System.out.println("         count      = " + mh.value.count);
            System.out.println("         text       = " + mh.value.text);
            System.out.println("SampleInfo.sample_rank = " +
                               sih.value.sample_rank);
          }
          else if (sih.value.instance_state ==
                     NOT_ALIVE_DISPOSED_INSTANCE_STATE.value) {
```

```
            System.out.println ("instance is disposed");
          }
          else if (sih.value.instance_state ==
                     NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.value) {
            System.out.println ("instance is unregistered");
          }
          else {
            System.out.println ("DataReaderListenerImpl::on_data_available: "+
                                "received unknown instance state "+
                                sih.value.instance_state);
          }

        } else if (status == RETCODE_NO_DATA.value) {
          System.err.println ("ERROR: reader received DDS::RETCODE_NO_DATA!");
        } else {
          System.err.println ("ERROR: read Message: Error: "+ status);
        }
    }

    .
    .
    .
}
```

The `SampleInfo` contains meta-information regarding the message such as the message validity, instance state, etc.

10.7 Cleaning up OpenDDS Java Clients

An application should clean up its OpenDDS environment with the following steps:

```
dp.delete_contained_entities();
```

Cleans up all topics, subscribers and publishers associated with that `Participant`.

```
dpf.delete_participant(dp);
```

The `DomainParticipantFactory` reclaims any resources associated with the `DomainParticipant`.

```
TheServiceParticipant.shutdown();
```

Shuts down the `ServiceParticipant`. This cleans up all OpenDDS associated resources. Cleaning up these resources is necessary to prevent the `DCPSInfoRepo` from forming associations between endpoints which no longer exist.

10.8 Configuring the Example

OpenDDS offers a file-based configuration mechanism. The syntax of the configuration file is similar to a Windows INI file. The properties are divided into named sections corresponding to common and individual transports configuration.

The Messenger example has common properties for the `DCPSInfoRepo` objects location and the global transport configuration:

```
[common]
DCPSInfoRepo=file://repo.ior
DCPSGlobalTransportConfig=$file
```

and a transport instance section with a transport type property:

```
[transport/1]
transport_type=tcp
```

The `[transport/1]` section contains configuration information for the transport instance named "1". It is defined to be of type `tcp`. The global transport configuration setting above causes this transport instance to be used by all readers and writers in the process.

See Chapter 7 for a complete description of all OpenDDS configuration parameters.

10.9 Running the Example

To run the Messenger Java OpenDDS application, use the following commands:

```
$DDS_ROOT/bin/DCPSInfoRepo -o repo.ior

$JAVA_HOME/bin/java -ea -cp classes:$DDS_ROOT/lib/i2jrt.jar:
$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes TestPublisher -DCPSConfigFile pub_tcp.ini

$JAVA_HOME/bin/java -ea -cp classes:$DDS_ROOT/lib/i2jrt.jar:
$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes TestSubscriber -DCPSConfigFile sub_tcp.ini
```

The `-DCPSConfigFile` command-line argument passes the location of the OpenDDS configuration file.

10.10 Java Message Service (JMS) Support

OpenDDS provides partial support for JMS version 1.1 <*http://java.sun.com/products/jms/*>. Enterprise Java applications can make use of the complete OpenDDS middleware just like standard Java and C++ applications.

See the `INSTALL` file in the `$DDS_ROOT/java/jms/` directory for information on getting started with the OpenDDS JMS support, including the prerequisites and dependencies.

CHAPTER 11

OpenDDS Modeling SDK

The OpenDDS Modeling SDK is a modeling tool that can be used by the application developer to define the required middleware components and data structures as a UML model and then generate the code to implement the model using OpenDDS. The generated code can then be compiled and linked with the application to provide seamless middleware support to the application.

11.1 Overview

11.1.1 Model Capture

UML models defining DCPS elements and policies along with data definitions are captured using the graphical model capture editors included in the Eclipse plug-ins. The elements of the UML models follow the structure of the DDS UML Platform Independent Model (PIM) defined in the DDS specification (OMG: `formal/07-01-01`).

Opening a new OpenDDS model within the plug-ins begins with a top level main diagram. This diagram includes any package structures to be included in the model along with the local QoS policy definitions, data definitions, and DCPS elements of the model. Zero or more of the policy or data definition elements can be included. Zero or one DCPS elements definition can be included in any given model.

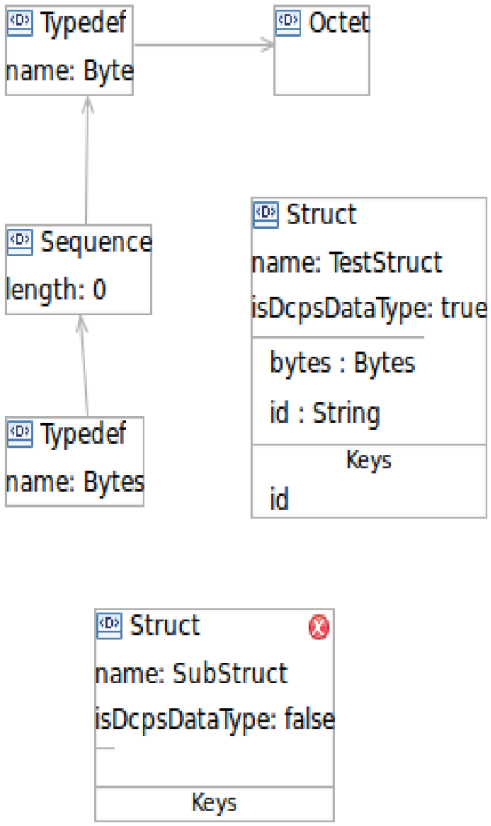


Figure 11-1 Graphical modeling of the data definitions

Creating separate models for QoS policies only, data definitions only, or DCPS elements only is supported. References to other models allows externally defined models to be included in a model. This allows sharing of data definitions and QoS policies among different DCPS models as well as including externally defined data in a new set of data definitions.

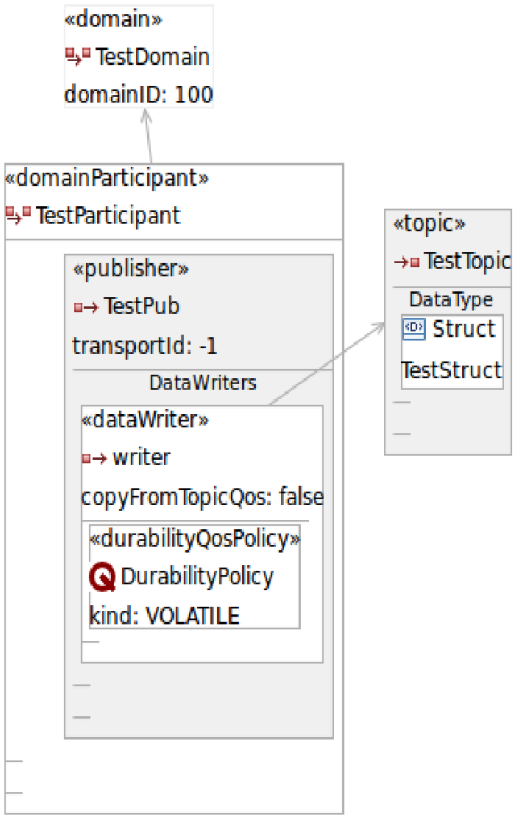


Figure 11-2 Graphical modeling of the DCPS entities

11.1.2 Code Generation

Once models have been captured, source code can be generated from them. This source code can then be compiled into link libraries providing the middleware elements defined in the model to applications that link the library. Code generation is done using a separate forms based editor.

Specifics of code generation are unique to the individual generation forms and are kept separate from the models for which generation is being performed. Code generation is performed on a single model at a time and includes the ability to tailor the generated code as well as specifying search paths to be used for locating resources at build time.

It is possible to generate model variations (distinct customizations of the same model) that can then be created within the same application or different applications. It is also possible to specify locations to search for header files and link libraries at build time.

See section 11.3.2 , “Generated Code” for details.

11.1.3 Programming

In order to use the middleware defined by models, applications need to link in the generated code. This is done through header files and link libraries. Support for building applications using the MPC portable build tool is included in the generated files for a model.

See section 11.3 , “Developing Applications” for details.

11.2 Installation and Getting Started

Unlike the OpenDDS Middleware which is compiled from source code by the developer, the compiled Modeling SDK is available for download via an Eclipse Update Site.

11.2.1 Prerequisites

- Java Runtime Environment (JRE)
  - Version 6 Update 24 (1.6.0_24) is current as of this writing
  - Download from *http://www.java.com*
- Eclipse IDE
  - Version 3.5 “Galileo”, service release 2 (3.5.2) is current as of this writing
  - Download from *http://www.eclipse.org/downloads/packages/release/galileo/sr2*

11.2.2 Installation

1) From Eclipse, open the Help menu and select Install New Software.

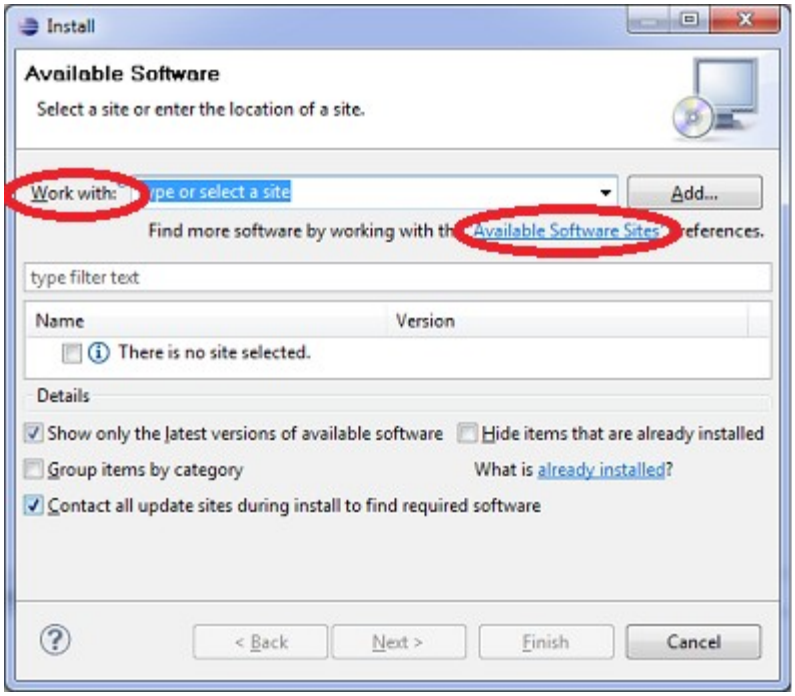


Figure 11-3 Eclipse Software Installation Dialog

9) Click the hyperlink for Available Software Sites.

10) The standard `eclipse.org` sites (Eclipse Project Updates and Galileo) should be enabled. If they are disabled, enable them now.

11) Add a new Site entry named OpenDDS with URL *http://www.opendds.org/modeling/eclipse*

12) Click OK to close the Preferences dialog and return to the Install dialog.

13) In the “Work with” combo box, select the new entry for OpenDDS.

14) Select the “OpenDDS Modeling SDK” and click Next.

15) Review the “Install Details” list and click Next. Review the license, select Accept (if you do accept it), and click Finish.

16) Eclipse will download the OpenDDS plug-ins and various plug-ins from `eclipse.org` that they depend on. There will be a security warning because the OpenDDS plug-ins are not signed. There also may be a prompt to accept a certificate from `eclipse.org`.

17) Eclipse will prompt the user to restart in order to use the newly installed software.

11.2.3 Getting Started

The OpenDDS Modeling SDK contains an Eclipse Perspective. Open it by going to the Window menu and selecting Open Perspective -> Other -> OpenDDS Modeling.

To get started using the OpenDDS Modeling SDK, see the help content installed in Eclipse. Start by going to the Help menu and selecting Help Contents. There is a top-level item for "*OpenDDS Modeling SDK Guide*" that contains all of the OpenDDS-specific content describing the modeling and code generation activities.

11.3 Developing Applications

In order to build an application using the OpenDDS Modeling SDK, one must understand a few key concepts. The concepts concern:

1) The support library
2) Generated model code
3) Application code

11.3.1 Modeling Support Library

The OpenDDS Modeling SDK includes a support library, found at *$DDS_ROOT/tools/modeling/codegen/model*. This support library, when combined with the code generated by the Modeling SDK, greatly reduces the amount of code needed to build an OpenDDS application.

The support library is a C++ library which is used by an OpenDDS Modeling SDK application. Two classes in the support library that most developers will need are the Application and Service classes.

11.3.1.1 The Application Class

The `OpenDDS::Model::Application` class takes care of initialization and finalization of the OpenDDS library. It is required for any application using OpenDDS to instantiate a single instance of the `Application` class, and further that the `Application` object not be destroyed while communicating using OpenDDS.

The `Application` class initializes the factory used to create OpenDDS participants. This factory requires the user-provided command line arguments. In order to provide them, the `Application` object must be provided the same command line arguments.

11.3.1.2 The Service Class

The `OpenDDS::Model::Service` class is responsible for the creation of OpenDDS entities described in an OpenDDS Modeling SDK model. Since the model can be generic, describing

a much broader domain than an individual application uses, the Service class uses lazy instantiation to create OpenDDS entities.

In order to properly instantiate these entities, it must know:

- The relationships among the entities
- The transport configuration used by entities

11.3.2 Generated Code

The OpenDDS Modeling SDK generates model-specific code for use by an OpenDDS Modeling SDK application. Starting with a .codegen file (which refers to an *.opendds* model file), the files described in Table 11-1. The process of generating code is documented in the Eclipse help.

Table 11-1 Generated Files

File Name	Description
<ModelName>.idl	Data types from the model's DataLib
<ModelName>_T.h	C++ class from the model's DcpsLib
<ModelName>_T.cpp	C++ implementation of the model's DcpsLib
<ModelName>.mpc	MPC project file for the generated C++ library
<ModelName>.mpb	MPC base project for use by the application
<ModelName>_paths.mpb	MPC base project with paths, see section 1.1.3.7
<ModelName>Traits.h	Transport configuration from the .codegen file
<ModelName>Traits.cpp	Transport configuration from the .codegen file

11.3.2.1 The DCPS Model Class

The DCPS library models relationships between DDS entities, including Topics, DomainParticipants, Publishers, Subscribers, DataWriters and DataReaders, and their corresponding Domains.

For each DCPS library in your model, the OpenDDS Modeling SDK generates a class named after the DCPS library. This DCPS model class is named after the DCPS library, and is found in the *<ModelName>_T.h* file in the code generation target directory.

The model class contains an inner class, named Elements, defining enumerated identifiers for each DCPS entity modeled in the library and each type referenced by the library's Topics. This Elements class contains enumeration definitions for each of:

- DomainParticipants
- Types
- Topics
- Content Filtered Topics
- Multi Topics
- Publishers
- Subscribers

- Data Writers
- Data Readers

In addition, the DCPS model class captures the relationships between these entities. These relationships are used by the Service class when instantiating DCPS entities.

11.3.2.2 The Traits Class

Entities in a DCPS model reference their transport configuration by name. The Model Customization tab of the Codegen file editor is used to define the transport configuration for each name.

There can be more than one set of configurations defined for a specific code generation file. These sets of configurations are grouped into instances, each identified by a name. Multiple instances may be defined, representing different deployment scenarios for models using the application.

For each of these instances, a `Traits` class is generated. The traits class provides the transport configuration modeled in the Codegen editor for a specific transport configuration name.

11.3.2.3 The Service Typedef

The Service is a template which needs two parameters: (1) the entity model, in the DCPS model `Elements` class, (2) transport configuration, in a `Traits` class. The OpenDDS Modeling SDK generates one typedef for each combination of DCPS library and transport configuration model instance. The typedef is named `<InstanceName><DCPSLibraryName>Type`.

11.3.2.4 Data Library Generated Code

From the data library, IDL is generated, which is processed by the IDL compilers. The IDL compilers generate type support code, which is used to serialize and deserialize data types.

11.3.2.5 QoS Policy Library Generated Code

There are no specific compilation units generated from the QoS policy library. Instead, the DCPS library stores the QoS policies of the entities it models. This QoS policy is later queried by the Service class, which sets the QoS policy upon entity creation.

11.3.3 Application Code Requirements

11.3.3.1 Required headers

The application will need to include the `Traits` header, in addition to the *`Tcp.h`* header (for static linking). These will include everything required to build a publishing application. Here is the `#include` section of an example publishing application, *`MinimalPublisher.cpp`*.

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"
```

11.3.3.2 Exception Handling

It is recommended that Modeling SDK applications catch both `CORBA::Exception` objects and `std::exception` objects.

```
int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    // Create and use OpenDDS Modeling SDK (see sections below)
  } catch (const CORBA::Exception& e) {
    // Handle exception and return non-zero
  } catch (const OpenDDS::DCPS::Transport::Exception& te) {
    // Handle exception and return non-zero
  } catch (const std::exception& ex) {
    // Handle exception and return non-zero
  }
  return 0;
}
```

11.3.3.3 Instantiation

As stated above, an OpenDDS Modeling SDK application must create an `OpenDDS::Model::Application` object for the duration of its lifetime. This `Application` object, in turn, is passed to the constructor of the Service object specified by one of the typedef declarations in the traits headers.

The service is then used to create OpenDDS entities. The specific entity to create is specified using one of the enumerated identifiers specified in the `Elements` class. The Service provides this interface for entity creation:

```
DDS::DomainParticipant_var participant(Elements::Participants::Values part);
DDS::TopicDescription_var topic(Elements::Participants::Values part,
                                Elements::Topics::Values       topic);
DDS::Publisher_var publisher(Elements::Publishers::Values publisher);
DDS::Subscriber_var subscriber(Elements::Subscribers::Values subscriber);
DDS::DataWriter_var writer(Elements::DataWriters::Values writer);
DDS::DataReader_var reader(Elements::DataReaders::Values reader);
```

It is important to note that the service also creates any required intermediate entities, such as `DomainParticipants`, `Publishers`, `Subscribers`, and `Topics`, when necessary.

11.3.3.4 Publisher Code

Using the `writer()` method shown above, *`MinimalPublisher.cpp`* continues:

```
int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    OpenDDS::Model::Application application(argc, argv);
    MinimalLib::DefaultMinimalType model(application, argc, argv);

    using OpenDDS::Model::MinimalLib::Elements;
    DDS::DataWriter_var writer = model.writer(Elements::DataWriters::writer);
```

What remains is to narrow the `DataWriter` to a type-specific data writer, and send samples.

```
    data1::MessageDataWriter_var msg_writer =
        data1::MessageDataWriter::_narrow(writer);
    data1::Message message;
    // Populate message and send
    message.text = "Worst. Movie. Ever.";
    DDS::ReturnCode_t error = msg_writer->write(message, DDS::HANDLE_NIL);
    if (error != DDS::RETCODE_OK) {
      // Handle error
    }
```

In total our publishing application, *`MinimalPublisher.cpp`*, looks like this:

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"

int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    OpenDDS::Model::Application application(argc, argv);
    MinimalLib::DefaultMinimalType model(application, argc, argv);

    using OpenDDS::Model::MinimalLib::Elements;
    DDS::DataWriter_var writer = model.writer(Elements::DataWriters::writer);

    data1::MessageDataWriter_var msg_writer =
        data1::MessageDataWriter::_narrow(writer);
    data1::Message message;
    // Populate message and send
    message.text = "Worst. Movie. Ever.";
    DDS::ReturnCode_t error = msg_writer->write(message, DDS::HANDLE_NIL);
    if (error != DDS::RETCODE_OK) {
      // Handle error
    }
  } catch (const CORBA::Exception& e) {
    // Handle exception and return non-zero
  } catch (const std::exception& ex) {
```

```
    // Handle exception and return non-zero
  }
  return 0;
}
```

Note this minimal example ignores logging and synchronization, which are issues that are not specific to the OpenDDS Modeling SDK.

11.3.3.5 Subscriber Code

The subscriber code is much like the publisher. For simplicity, OpenDDS Modeling SDK subscribers may want to take advantage of a base class for Reader Listeners, called `OpenDDS::Modeling::NullReaderListener`. The `NullReaderListener` implements the entire `DataReaderListener` interface and logs every callback.

Subscribers can create a listener by deriving a class from `NullReaderListener` and overriding the interfaces of interest, for example on_data_available.

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"
#include <model/NullReaderListener.h>

class ReaderListener : public OpenDDS::Model::NullReaderListener {
public:
  virtual void on_data_available(DDS::DataReader_ptr reader)
                       ACE_THROW_SPEC((CORBA::SystemException)) {
    data1::MessageDataReader_var reader_i =
      data1::MessageDataReader::_narrow(reader);

    if (!reader_i) {
      // Handle error
      ACE_OS::exit(-1);
    }

    data1::Message msg;
    DDS::SampleInfo info;

    // Read until no more messages
    while (true) {
      DDS::ReturnCode_t error = reader_i->take_next_sample(msg, info);
      if (error == DDS::RETCODE_OK) {
        if (info.valid_data) {
          std::cout << "Message: " << msg.text.in() << std::endl;
        }
      } else {
        if (error != DDS::RETCODE_NO_DATA) {
          // Handle error
        }
        break;
      }
    }
  }
};
```

In the main function, create a data reader from the service object:

```
DDS::DataReader_var reader = model.reader(Elements::DataReaders::reader);
```

Naturally, the `DataReaderListener` must be associated with the data reader in order to get its callbacks.

```
DDS::DataReaderListener_var listener(new ReaderListener);
reader->set_listener(listener, OpenDDS::DCPS::DEFAULT_STATUS_MASK);
```

The remaining subscriber code has the same requirements of any OpenDDS Modeling SDK application, in that it must initialize the OpenDDS library through an `OpenDDS::Modeling::Application` object, and create a Service object with the proper DCPS model Elements class and traits class.

An example subscribing application, *`MinimalSubscriber.cpp`*, follows.

```
#ifdef ACE_AS_STATIC_LIBS
#include <dds/DCPS/transport/tcp/Tcp.h>
#endif

#include "model/MinimalTraits.h"
#include <model/NullReaderListener.h>

class ReaderListener : public OpenDDS::Model::NullReaderListener {
public:
  virtual void on_data_available(DDS::DataReader_ptr reader)
                       ACE_THROW_SPEC((CORBA::SystemException)) {
    data1::MessageDataReader_var reader_i =
      data1::MessageDataReader::_narrow(reader);

    if (!reader_i) {
      // Handle error
      ACE_OS::exit(-1);
    }

    data1::Message msg;
    DDS::SampleInfo info;

    // Read until no more messages
    while (true) {
      DDS::ReturnCode_t error = reader_i->take_next_sample(msg, info);
      if (error == DDS::RETCODE_OK) {
        if (info.valid_data) {
          std::cout << "Message: " << msg.text.in() << std::endl;
        }
      } else {
        if (error != DDS::RETCODE_NO_DATA) {
          // Handle error
        }
        break;
      }
    }
  }
};
```

```
int ACE_TMAIN(int argc, ACE_TCHAR* argv[])
{
  try {
    OpenDDS::Model::Application application(argc, argv);
    MinimalLib::DefaultMinimalType model(application, argc, argv);

    using OpenDDS::Model::MinimalLib::Elements;

    DDS::DataReader_var reader = model.reader(Elements::DataReaders::reader);

    DDS::DataReaderListener_var listener(new ReaderListener);
    reader->set_listener(listener, OpenDDS::DCPS::DEFAULT_STATUS_MASK);

    // Call on_data_available in case there are samples which are waiting
    listener->on_data_available(reader);

    // At this point the application can wait for an exteral “stop” indication
    // such as blocking until the user terminates the program with Ctrl-C.

  } catch (const CORBA::Exception& e) {
    e._tao_print_exception("Exception caught in main():");
    return -1;
  } catch (const std::exception& ex) {
    // Handle error
    return -1;
  }
  return 0;
}
```

11.3.3.6 MPC Projects

In order to make use of the OpenDDS Modeling SDK support library, OpenDDS Modeling SDK MPC projects should inherit from the dds_model project base. This is in addition to the dcpsexe base from which non-Modeling SDK projects inherit.

```
project(*Publisher) : dcpsexe, dds_model {
  // project configuration
}
```

The generated model library will generate an MPC project file and base project file in the target directory, and take care of building the model shared library. OpenDDS modeling applications must both (1) include the generated model library in their build and (2) ensure their projects are built after the generated model libraries.

```
project(*Publisher) : dcpsexe, dds_model {
  // project configuration
  libs  += Minimal
  after += Minimal
}
```

Both of these can be accomplished by inheriting from the model library's project base, named after the model library.

```
project(*Publisher) : dcpsexe, dds_model, Minimal {
  // project configuration
```

```
}
```

Note that the *Minimal.mpb* file must now be found by MPC during project file creation. This can be accomplished through the -include command line option.

Using either form, the MPC file must tell the build system where to look for the generated model library.

```
project(*Publisher) : dcpsexe, dds_model, Minimal {
  // project configuration
  libpaths += model
}
```

This setting based upon what was provided to the Target Folder setting in the Codegen file editor.

Finally, like any other MPC project, its source files must be included:

```
  Source_Files {
    MinimalPublisher.cpp
  }
```

The final MPC project looks like this for the publisher:

```
project(*Publisher) : dcpsexe, dds_model, Minimal {
  exename   = publisher
  libpaths += model

  Source_Files {
    MinimalPublisher.cpp
  }
}
```

And similar for the subscriber:

```
project(*Subscriber) : dcpsexe, dds_model, Minimal {
  exename   = subscriber
  libpaths += model

  Source_Files {
    MinimalSubscriber.cpp
  }
}
```

11.3.3.7 Dependencies Between Models

One final consideration -- the generated model library could itself depend on other generated model libraries. For example, there could be an external data type library which is generated to a different directory.

This possibility could cause a great deal of maintenance of project files, as models change their dependencies over time. To help overcome this burden, the generated model library

records the paths to all of its externally referenced model libraries in a separate MPB file named *<ModelName>_paths.mpb*. Inheriting from this paths base project will inherit the needed settings to include the dependent model as well.

Our full MPC file looks like this:

```
project(*Publisher) : dcpsexe, dds_model, Minimal, Minimal_paths {
  exename   = publisher
  libpaths += model

  Source_Files {
    MinimalPublisher.cpp
  }
}

project(*Subscriber) : dcpsexe, dds_model, Minimal, Minimal_paths {
  exename   = subscriber
  libpaths += model

  Source_Files {
    MinimalSubscriber.cpp
  }
}
```

CHAPTER 12

OpenDDS Recorder and Replayer

12.1 Overview

The Recorder feature of OpenDDS allows applications to record samples published on arbitrary topics without any prior knowledge of the data type used by that topic. Analogously, the Replayer feature allows these recorded samples to by re-published back into the same or other topics. What makes these features different from other Data Readers and Writers are their ability to work with any data type, even if unknown at application build time. Effectively, the samples are treated as an opaque byte sequence.

The purpose of this chapter is to describe the public API for OpenDDS to enable the recording/replaying use-case.

12.2 API Structure

Two new user-visible classes (that behave somewhat like their DDS Entity counterparts) are defined in the `OpenDDS::DCPS` namespace, along with the associated Listener interfaces. Listeners may be optionally implemented by the application. The `Recorder` class acts similarly to a `DataReader` and the `Replayer` class acts similarly to a `DataWriter`.

Both `Recorder` and `Replayer` make use of the underlying OpenDDS discovery and transport libraries as if they were `DataReader` and `DataWriter`, respectively. Regular OpenDDS

applications in the domain will “see” the Recorder objects as if they were remote DataReaders and Replayers as if they were DataWriters.

12.3 Usage Model

The application creates any number of Recorders and Replayers as necessary. This could be based on using the Built-In Topics to dynamically discover which topics are active in the Domain. Creating a Recorder or Replayer requires the application to provide a topic name and type name (as in DomainParticipant::create_topic()) and also the relevant QoS data structures. The Recorder requires the SubscriberQos and DataReaderQos and the Replayer requires the PublisherQos and DataWriterQos. These values are used in discovery's reader/writer matching. See the section on QoS processing below for how the Recorder and Replayer use QoS. Here is the code needed to create a recorder:

```
 OpenDDS::DCPS::Recorder_rch recorder =
      service_participant->create_recorder(domain_participant,
                                           topic.in(),
                                           sub_qos,
                                           dr_qos,
                                           recorder_listener);
```

Data samples are made available to the application via the RecorderListener using a simple “one callback per sample” model. The sample is delivered as an OpenDDS::DCPS::RawDataSample object. This object includes the timestamp for that data sample as well as the marshaled sample value. Here is a class definition for a user-defined Recorder Listener.

```
class MessengerRecorderListener : public OpenDDS::DCPS::RecorderListener
{
public:
  MessengerRecorderListener();

  virtual void on_sample_data_received(OpenDDS::DCPS::Recorder*,
                                       const OpenDDS::DCPS::RawDataSample& sample);

  virtual void on_recorder_matched(OpenDDS::DCPS::Recorder*,
                                   const ::DDS::SubscriptionMatchedStatus& status );

};
```

The application can store the data wherever it sees fit (in memory, file system, database, etc.). At any later time, the application can provide that same sample to a Replayer object configured for the same topic. It’s the application’s responsibility to make sure the topic types match. Here is a sample call that replays a sample to all readers connected on a replayer’s topic:

```
     replayer->write(sample);
```

Because the stored data is dependent on the definition of the data structure, it can't be used across different versions of OpenDDS or different versions of the IDL used by the OpenDDS participants.

12.4 QoS Processing

The lack of detailed knowledge about the data sample complicates the use of many normal DDS QoS properties on the `Replayer` side. The properties can be divided into a few categories:

- Supported
  - Liveliness
  - Time-Based Filter
  - Lifespan
  - Durability (transient local level, see details below)
  - Presentation (topic level only)
  - Transport Priority (pass-thru to transport)
- Unsupported
  - Deadline (still used for reader/writer match)
  - History
  - Resource Limits
  - Durability Service
  - Ownership and Ownership Strength (still used for reader/writer match)
- Affects reader/writer matching and Built-In Topics but otherwise ignored
  - Partition
  - Reliability (still used by transport negotiation)
  - Destination Order
  - Latency Budget
  - User/Group Data

12.4.1 Durability Details

On the `Recorder` side, transient local durability works just the same as any normal `DataReader`. Durable data is received from matched `DataWriters`. On the `Replayer` side there are some differences. As opposed to the normal DDS `DataWriter`, `Replayer` is not caching/storing any data samples (they are simply sent to the transport). Because instances are not known, storing data samples according to the usual History and Resource Limits rules is not possible. Instead, transient local durability can be supported with a "pull" model whereby the middleware invokes a method on the `ReplayerListener` when a new remote

`DataReader` is discovered. The application can then call a method on the `Replayer` with any data samples that should be sent to that newly-joined `DataReader`. Determining which samples these are is left to the application.