OpenDDS Developer's Guide

OpenDDS Version 2.0.1

(Document Revision 1)
Supported by Object Computing, Inc. (OCI)
http://www.opendds.org
http://www.ociweb.com

Contents

Contents . . . . **iii**

Preface . . . . **ix**

Chapter 1 **Introduction** 1

DCPS Overview 2

Basic Concepts 2

Built-In Topics 5

Quality of Service Policies 5

Listeners 6

Conditions 6

OpenDDS Implementation 7

Compliance 7

OpenDDS Architecture 11

Chapter 2 **Getting Started** 15

Using DCPS 15

Defining the Data Types 15

Processing the IDL 16
Starting the DCPS Information Repository 19
A Simple Message Publisher 19
Setting up the Subscriber 25
The Data Reader Listener Implementation 28
Cleaning up in OpenDDS Clients 30
Running the Example 31
Data Handling Optimizations 32
Registering and Using Instances in the Publisher 32
Reading Multiple Samples 33
Zero-Copy Read 33

Chapter 3 Quality of Service 37
Introduction 37
Supported Policies 37
Default QoS Policy Values 39
LIVELINESS 43
RELIABILITY 44
HISTORY 45
DURABILITY 46
DURABILITY_SERVICE 47
RESOURCE_LIMITS 48
PARTITION 49
DEADLINE 49
LIFESPAN 50
USER_DATA 51
TOPIC_DATA 51
GROUP_DATA 51
TRANSPORT_PRIORITY 52
LATENCY_BUDGET 53
ENTITY_FACTORY 55
PRESENTATION 56
DESTINATION_ORDER 57
WRITER_DATA_LIFECYCLE 58
READER_DATA_LIFECYCLE 59
TIME_BASED_FILTER 59

Unsupported Policies 60
OWNERSHIP 60
OWNERSHIP_STRENGTH 61
Policy Example 61

Chapter 4 Conditions and Listeners 63
Introduction 63
Communication Status Types 64
Topic Status Types 64
Subscriber Status Types 65
Data Reader Status Types 65
Data Writer Status Types 68
Listeners 70
Topic Listener 72
Data Writer Listener 72
Publisher Listener 72
Data Reader Listener 73
Subscriber Listener 73
Domain Participant Listener 73
Conditions 73
Overview 73
Status Condition Example 74
Additional Condition Types 74
Read Conditions 75
Query Conditions 75
Guard Conditions 75

Chapter 5 Configuration 77
Configuration Files 77
Common Configuration Settings 78
Transport Configuration Settings 81
Multiple DCPSInfoRepo Configuration 88
Logging 89
DCPS Layer Logging 89
Transport Layer Logging 90

Chapter 6 Pluggable Transports 91
Simple TCP Transport 91
Unreliable Datagram Transports 93
SimpleUDP Transport 93
SimpleMcast Transport 94
Reliable Multicast Transport 95

Chapter 7 Built-In Topics 97
Introduction 97
Building Without BIT Support 98
DCPSParticipant Topic 98
DCPSTopic Topic 99
DCPSPublication Topic 99
DCPSSubscription Topic 100
Built-In Topic Subscription Example 101

Chapter 8 dcps_ts.pl Options 103
dcps_ts.pl Command Line Options 103

Chapter 9 The DCPS Information Repository 107
DCPS Information Repository Options 107
Repository Federation 109
Federation Management 111
Federation Example 113

Chapter 10 OpenDDS Java Bindings 117
Introduction 117
IDL and Code Generation 117
Setting up an OpenDDS Java Project 119
A Simple Message Publisher 121
Initializing the Participant 121
Registering the Data Type and Creating a Topic 122
Initializing and Registering the Transport 123
Creating a Publisher 123

Creating a DataWriter and Registering an Instance 123
Setting up the Subscriber 124
Creating a Subscriber 125
Creating a DataReader and Listener 125
The DataReader Listener Implementation 125
Cleaning up OpenDDS Java Clients 127
Configuring the Example 128
Running the Example 128
Java Message Service (JMS) Support 129

Index 131

OBJECT COMPUTING, INC.

Preface

What Is OpenDDS?

OpenDDS is an open source implementation of the OMG Data Distribution Service (DDS) for Real-Time Systems specification (OMG Document formal/07-01-01). 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 2.0.1. 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 2.0 Release

OpenDDS version 2.0 includes many new features and improvements over the previous release. This section highlights some of the more important and visible changes. See the release notes in the distribution ($DDS_ROOT/DDS_release_notes.txt) and the appropriate sections of this guide for more details on these and other features.

DDS Specification Compliance

With this release, OpenDDS is fully compliant with the Minimum and Persistence profiles of the DDS version 1.2 specification (OMG Document formal/07-01-01).

Examples of changes in this release that affect compliance with the DDS specification include:

- Implementations of many quality of service (QoS) policies have been added or improved, including:
    - PRESENTATION—All settings of coherent access and ordered access are now supported for the `INSTANCE` and `TOPIC` access scope settings; `GROUP` access scope (required for DCPS support for the Object Model profile) is not yet supported;
    - LIVELINESS—`MANUAL_BY_TOPIC` and `MANUAL_BY_PARTICIPANT` liveliness kinds are now supported in addition to `AUTOMATIC`;
    - DESTINATION_ORDER—`BY_SOURCE_TIMESTAMP` ordering of samples is now supported, as well as `BY_RECEPTION_TIMESTAMP`; in addition, the implementation of DESTINATION_ORDER is consistent with the ordered access setting of the PRESENTATION policy;
    - ENTITY_FACTORY—Newly created entities can now be enabled manually or automatically via this policy;
    - WRITER_DATA_LIFECYCLE—Data instances can now be disposed automatically upon unregistration (including as a consequence of data writer deletion) via this policy;
    - READER_DATA_LIFECYCLE—Data samples maintained by a data reader can now be purged automatically and resources allocated for them released, after a specified duration, when either no writers are alive or the data instances for those samples have been disposed, according to the settings of this policy;
    - TIME_BASED_FILTER—Data readers can now use this policy to control the minimum separation time between samples, on a per data instance basis, independently of the rate at which the samples are written by the associated data writer;
- APIs have been updated to comply with the DDS version 1.2 specification, including changes to constant, type, and structure names, as well as new or modified operation names and signatures on some interfaces.

See 1.3.1 for further details of OpenDDS's compliance with the DDS specification.

Other Implementation Improvements

- The implementation of instance handles has been improved. In particular, unregistering and disposing instances now results in proper release of resources once all samples in an affected instance have been removed.

TAO Version Compatibility

- OpenDDS 2.0 is compatible with the current patch levels of TAO 1.5a and 1.6a, as well as the current DOC Group beta/micro release. TAO 1.4a is no longer supported as of this release. See the `$DDS_ROOT/README` file for details on TAO versions that are compatible with OpenDDS.

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.

OMG Specification References

Throughout this guide, we refer to various specifications published by the Object Management Group (OMG). These 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 Documents

Additional documentation on OpenDDS is available from the *OpenDDS Community Portal* at `<http://www.opendds.org>`. In particular, be sure to see the build instructions, architectural overview, Doxygen-generated reference pages, and other information at `<http://www.opendds.org/documentation.html>`, and visit the OpenDDS Frequently Asked Questions (FAQ) pages at `<http://www.opendds.org/faq.html>`.

Additional information and documents about DDS are available from the *OMG Data Distribution Portal* at `<http://portals.omg.org/dds>`.

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 and JacORB), J2EE (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 <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, 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 CORBAservices 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 three-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.

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.

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 (OMG Document formal/07-01-01). OpenDDS is sponsored by Object Computing, Inc. (OCI) and is available via `http://www.opendds.org/`. This developer's guide is based on the version 2.0.1 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.

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 recent patch levels of TAO version 1.5a, 1.6a, and 1.7.x. OpenDDS version 1.3 was the last release that supported TAO version 1.4a.

Note *OpenDDS currently implements a subset of 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 1.2.1 or at* `http://www.opendds.org/` *for more information.*

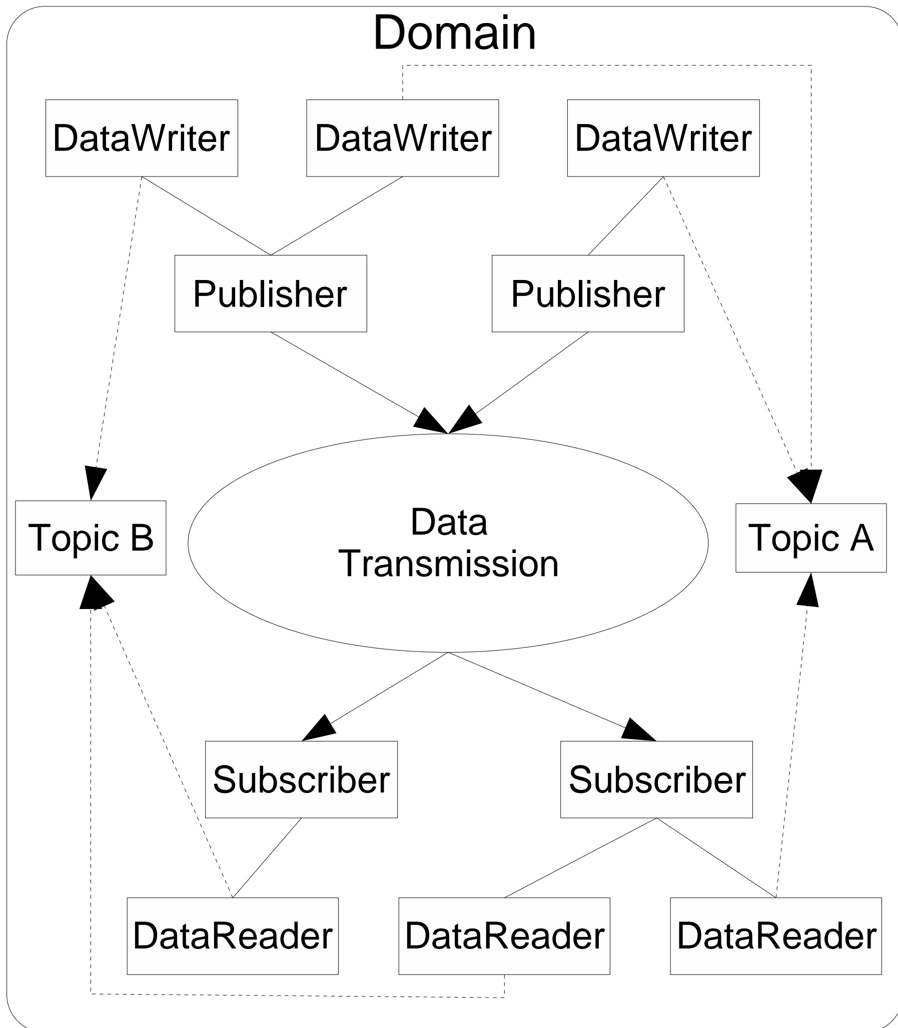
1.1 DCPS Overview

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:

Topic Name	Description
DCPSParticipant	Each instance represents a domain participant.
DCPSTopic	Each topic is an instance.
DCPSPublication	Each instance represents a data writer
DCPSSubscription	Each instance represents a data reader.

Figure 1-2 Built-In Topics

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 collaborate to specify QoS through an offer-request paradigm. Publishers *offer* a set of QoS policies to all subscribers. Subscribers *request* a set of policies that they require. The DDS implementation then attempts to match the requested policies with the offered policies. If the policies are consistent the subscription is initiated. If the policies are not consistent then the subscription attempt fails.

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

1.1.4 Listeners

The DPCS 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 Status Condition, and attaches it to a Wait Set.
- The application waits on the Wait Set until one or more Conditions become true.
- The application calls operations on the corresponding entity objects to extract the necessary information.
- The Data Reader interface also has operations that take a ReadCondition argument.
- Query Conditions with queries of the form `"ORDER BY ..."` are supported. These conditions are commonly used with the Data Reader interface and not in conjunction with Wait Sets.

Object Computing, Inc.

1.2 OpenDDS Implementation

1.2.1 Compliance

Appendix A 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

This section describes OpenDDS's compliance with these profiles in terms of the entities and quality of service policies defined by the DDS specification.

1.2.1.1 Entity Compliance

The DDS specification defines five modules that make up the DCPS PIM:

1. Infrastructure Module
2. Domain Module
3. Topic-Definition Module
4. Publication Module
5. Subscription Module

Various entities are defined within each module. Not all entities pertain to every profile listed in 1.2.1. Table 1-1 through Table 1-5 show which entities are included in each module and to which profiles each entity pertains, as well as whether or not the entity is implemented by OpenDDS.

Table 1-1 Infrastructure Module Entities

Entity Name	Profiles	Impl?
`Entity`	All	Yes
`DomainEntity`	All	Yes
`QosPolicy`	All	Yes
`Listener`	All	Yes
`Status`	All	Yes
`WaitSet`	All	Yes

Table 1-1 Infrastructure Module Entities

Entity Name	Profiles	Impl?
`Condition`	All	Yes
`GuardCondition`	All	Yes
`StatusCondition`	All	Yes

Table 1-2 Domain Module Entities

Entity Name	Profiles	Impl?
`DomainParticipant`	All	Yes
`DomainParticipantFactory`	All	Yes
`DomainParticipantListener`	All	Yes

Table 1-3 Topic-Definition Module Entities

Entity Name	Profiles	Impl?
`TopicDescription`	All	Yes
`Topic`	All	Yes
`ContentFilteredTopic`	Content-Subscription	No
`MultiTopic`	Content-Subscription	No
`TopicListener`	All	Yes
`TypeSupport`	All	Yes

Table 1-4 Publication Module Entities

Entity Name	Profiles	Impl?
`Publisher`	All	Yes
`DataWriter`	All	Yes
`PublisherListener`	All	Yes
`DataWriterListener`	All	Yes

Table 1-5 Subscription Module Entities

Entity Name	Profiles	Impl?
`Subscriber`	All	Yes
`DataReader`	All	Yes
`DataSample`	All	Yes
`SampleInfo`	All	Yes
`SubscriberListener`	All	Yes
`DataReaderListener`	All	Yes
`ReadCondition`	All	Yes
`QueryCondition`	Content-Subscription	Partial[1]

1. Only queries of the form "`ORDER BY ...`" are supported.

1.2.1.2 Quality of Service (QoS) Compliance

The DDS specification defines several QoS policies. Each policy is applicable to certain entities. Not all policies pertain to every profile listed in 1.2.1. Table 1-6 shows the various QoS policies and their possible values, the entities to which the policies apply, the profiles to which each policy/value pertains, as well as whether or not the policy/value is implemented by OpenDDS.

Table 1-6 QoS Policies

Policy Name	Entities	Values	Profiles	Impl?

`USER_DATA`	`DomainParticipant`
	`DataWriter`
	`DataReader`
		sequence of octets	All	Yes

`TOPIC_DATA`	`Topic`	sequence of octets	All	Yes
`GROUP_DATA`	`Publisher` `Subscriber`	sequence of octets	All	Yes

`DURABILITY`	`Topic`
	`DataWriter`
	`DataReader`
		`VOLATILE`	All	Yes

		`TRANSIENT_LOCAL`	All	Yes
		`TRANSIENT (includes DURABILITY_SERVICE)`	Persistence	Yes
		`PERSISTENT (includes DURABILITY_SERVICE)`	Persistence	Yes

Table 1-6 QoS Policies

Policy Name	Entities	Values	Profiles	Impl?
PRESENTATION	Publisher Subscriber	INSTANCE scope COHERENT=true	All	Yes
		INSTANCE scope ORDERED=true	All	Yes
		TOPIC scope COHERENT=true	All	Yes
		TOPIC scope ORDERED=true	All	Yes
		GROUP scope COHERENT=true	Object Model	No
		GROUP scope ORDERED=true	Object Model	No
DEADLINE	Topic DataWriter DataReader	integer (period)	All	Yes
LATENCY_BUDGET	Topic DataWriter DataReader	integer (duration)	All	Yes
OWNERSHIP	Topic DataWriter DataReader	SHARED	All	Yes
		EXCLUSIVE	Ownership	No
OWNERSHIP_STRENGTH	Topic DataWriter DataReader	integer (value)	Ownership	No
LIVELINESS	Topic DataWriter DataReader	AUTOMATIC	All	Yes
		MANUAL_BY_PARTICIPANT	All	Yes
		MANUAL_BY_TOPIC	All	Yes
TIME_BASED_FILTER	DataReader	integer (minimum_separation)	All	Yes
PARTITION	Publisher Subscriber	sequence of strings	All	Yes
RELIABILITY	Topic DataWriter[1] DataReader	BEST_EFFORT	All	Yes
		RELIABLE	All	Yes[2]
TRANSPORT_PRIORITY	Topic DataWriter	integer	All	Yes[3]
LIFESPAN	Topic DataWriter	integer (duration)	All	Yes
DESTINATION_ORDER	Topic DataWriter DataReader	BY_RECEPTION_TIMESTAMP	All	Yes
		BY_SOURCE_TIMESTAMP	All	Yes
HISTORY	Topic DataWriter DataReader	KEEP_LAST integer (depth)	All[4]	Yes
		KEEP_ALL	All	Yes

OBJECT COMPUTING, INC.

Table 1-6 QoS Policies

Policy Name	Entities	Values	Profiles	Impl?
RESOURCE_LIMITS	Topic DataWriter DataReader	integer (max_samples) integer (max_instances) integer (max_samples_per_instance)	All	Yes
ENTITY_FACTORY	DomainParticipantFactory DomainParticipant Publisher Subscriber	AUTO_ENABLE=true	All	Yes
		AUTO_ENABLE=false	All	Yes
WRITER_DATA_ LIFECYCLE	DataWriter	boolean (autodispose_unregistered_ instances)	All	Yes
READER_DATA_ LIFECYCLE	DataReader	integer (autopurge_nowriter_samples_ delay) integer (autopurge_disposed_samples_ delay)	All	Yes

1. 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 with the DDS specification).
2. `RELIABILITY.kind=RELIABLE` supported only if the TCP or Reliable Multicast transport implementation is used.
3. Not implemented as changeable.
4. `KEEP_LAST.depth > 1` only applies to the Ownership profile.

1.2.2 OpenDDS Architecture

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

1.2.2.1 Basic Philosophy

The OpenDDS implementation is based on a fairly strict interpretation of the OMG IDL Platform Specific Model (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 of things like listeners.

1.2.2.2 Pluggable Transport Layer

OpenDDS uses the CORBA interfaces defined by the DDS specification to initialize and control service usage. Data transmission is accomplished via an OpenDDS-specific *Pluggable Transport* layer that allows the service to be used with a variety of transport protocols. OpenDDS currently implements simple TCP, UDP, reliable multicast and unreliable multicast transports. Transports are created via a factory object and are associated with publishers and subscribers who use them for their data transmission.

The pluggable transport layer enables application developers to implement their own customized protocols. Implementing your own custom transport involves specializing a number of classes defined in the transport framework directory `$DDS_ROOT/dds/DCPS/transport/framework`. See the simple TCP implementation in `$DDS_ROOT/dds/DCPS/transport/simpleTCP` for details.

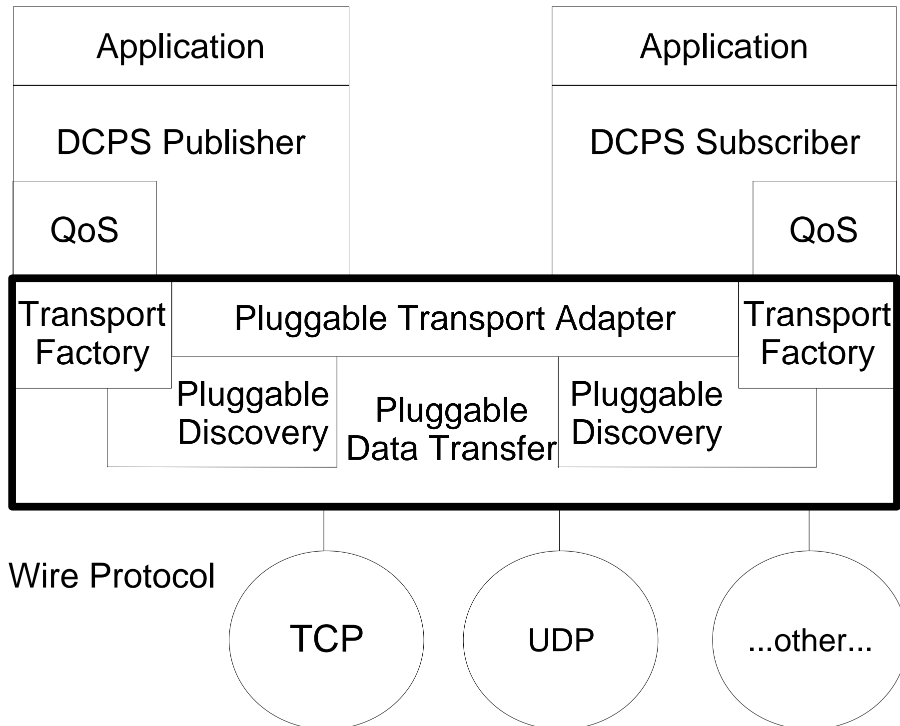


Figure 1-3 OpenDDS Pluggable Transport Framework

1.2.2.3 Custom Marshaling

Because data transmission is not done with CORBA, DDS implementations are free to marshal the data using customized formats. OpenDDS uses a more efficient variation of CORBA's Common Data Representation (CDR). A new IDL compiler switch (`-Gdcps`) causes the TAO IDL compiler to generate the appropriate marshaling and instance key support code for DCPS-enabled types.

1.2.2.4 DCPS Information Repository

The DCPS Information Repository (InfoRepo) acts as the intermediary or broker between the publisher and subscriber. It is currently 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 InfoRepo process needs to be running whenever OpenDDS is being used. The InfoRepo is not involved in data propagation, its role is limited in scope to publishers and subscribers discovering one another.

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.

1.2.2.5 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 the service thread and queued for reading by the application. Data reader listeners are called from the service thread.

1.2.2.6 Configuration

OpenDDS includes a file-based configuration framework for configuring both global items such as debug level, memory allocation, and `DCPSInfoRepo` locations, as well as transport implementations for publishers and subscribers. The complete set of configuration settings is described in Chapter 5.

CHAPTER 2

Getting Started

2.1 Using DCPS

This chapter focuses on an example application using DCPS to distribute data from a publisher process to a subscriber. 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 Simple TCP transport. Full source code for this example is in the OpenDDS source code distribution in the directory `$DDS_ROOT/DevGuideExamples/DCPS/Messenger`. 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` statements to identify the data types that DDS transmits and processes. These data types are processed by the TAO IDL compiler and the `dcps_ts.pl` script to generate code necessary for transmitting these types with DDS. 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. Currently, 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 the different instances within a topic that use this type. 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 name and the member name that is the key for that type. Multiple keys are specified via separate `DCPS_DATA_KEY` pragmas with the same data type. In the above example, we identify the `subject_id` member of `Messenger::Message` as the key. Each sample published with a unique `subject_id` value is defined as a different instance within the topic. 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 processed like any other IDL with the exception that we pass the `-Gdcps` option the TAO IDL compiler.

```
tao_idl -Gdcps Messenger.idl
```

1.Other types, such as structures, sequences, and arrays cannot be used directly as keys, though a work around is to declare (via the `DCPS_DATA_KEY` pragma) individual members of structs or elements of sequences/arrays as keys.

This causes the IDL compiler to generate additional serialization and key support code that OpenDDS uses to marshal and demarshal the `Message` structure.

In addition, we need to process the IDL file with the `dcps_ts.pl` script to generate the required type support code for the data readers and writers. This script 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`
- `<filename>TypeSupportImpl.h`
- `<filename>TypeSupportImpl.cpp`

For example, running `dcps_ts.pl` as follows

```
dcps_ts.pl 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 to generate stubs and skeletons. These and the implementation file should be linked with your OpenDDS applications that use the `Message` type. This type support generation script has a number of options that specialize the generated code. These options are described in Chapter 8.

Typically, you do not directly invoke the IDL compiler or `dcps_ts.pl` script 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 -Gdcps IDL compiler option and 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 dcps_ts.pl script. 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
  }
}
```


Object Computing, Inc.

2.1.3 Starting the DCPS Information Repository

The source code for the DCPS Information Repository server is found in `$DDS_ROOT/dds/InfoRepo` and the server executable is `$DDS_ROOT/bin/DCPSInfoRepo`. This server process hosts the DCPSInfo CORBA object that is the entry point for all OpenDDS functionality. This object is mapped against the key string ‘`DCPSInfoRepo`’ in the process’ IORTable. Thus a `corbaloc` ObjectURL such as:

```
corbaloc:iiop:localhost:12345/DCPSInfoRepo
```

can be used to locate the DCPSInfo object. The server also writes out the DCPSInfo object’s IOR as a string to a file, which can also be used to bootstrap clients. We can alter the file name used for writing this IOR with the `-o` command line option.

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

The full set of command line options for the DCPSInfoRepo server are documented in Chapter 9.

2.1.4 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`.

2.1.4.1 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,
```

```
                           DDS::DomainParticipantListener::_nil(),
                           OpenDDS::DCPS::DEFAULT_STATUS_MASK);
  if (CORBA::is_nil(participant.in())) {
    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 5. 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).

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

2.1.4.2 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.in (), "")) {
  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.in (),
                              TOPIC_QOS_DEFAULT,
                              DDS::TopicListener::_nil(),
                              OpenDDS::DCPS::DEFAULT_STATUS_MASK);
 if (CORBA::is_nil(topic.in())) {
   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.

2.1.4.3 Initializing and Registering the Transport

We now initialize the transport we want to use.

```
OpenDDS::DCPS::TransportImpl_rch transport_impl =
  TheTransportFactory->create_transport_impl (
    OpenDDS::DCPS::DEFAULT_SIMPLE_TCP_ID,
    OpenDDS::DCPS::AUTO_CONFIG);
```

This code obtains the transport implementation from the singleton transport factory, called `TheTransportFactory`. The `OpenDDS::DCPS::AUTO_CONFIG` argument indicates that we are using a configuration file to configure the transport implementation.The `OpenDDS::DCPS::DEFAULT_SIMPLE_TCP_ID` specifies the transport id value. Note that the code itself does not need to know any details about the transport implementation, such as whether it uses TCP or UDP, what its endpoints are, etc.

The code above uses the default simple TCP transport identity `DEFAULT_SIMPLE_TCP_ID`. OpenDDS reserves a range `(0xFFFFFF00 ~ 0xFFFFFFFF)` for default transport identities. Currently, only the simple TCP, simple UDP, simple multicast, and reliable multicast transport identifiers are supported. The default transport identifiers are defined in `TransportDef.h` as follows:

```
const TransportIdType DEFAULT_SIMPLE_TCP_ID          = 0xFFFFFF00;
const TransportIdType DEFAULT_SIMPLE_UDP_ID          = 0xFFFFFF01;
const TransportIdType DEFAULT_SIMPLE_MCAST_PUB_ID    = 0xFFFFFF02;
```

```
const TransportIdType DEFAULT_SIMPLE_MCAST_SUB_ID       = 0xFFFFFF03;
const TransportIdType DEFAULT_RELIABLE_MULTICAST_PUB_ID = 0xFFFFFF04;
const TransportIdType DEFAULT_RELIABLE_MULTICAST_SUB_ID = 0xFFFFFF05;
```

Alternatively, you can define your own transport identifier and specify the details of your transport via a configuration file. This is discussed in 5.1.2.

The `TransportFactory` also provides alternate APIs to create a transport implementation.

```
OpenDDS::DCPS::TransportIdType transport_impl_id = 1;
OpenDDS::DCPS::TransportImpl_rch transport_impl =
  TheTransportFactory->create_transport_impl (
    transport_impl_id, "SimpleTcp", OpenDDS::DCPS::AUTO_CONFIG);
```

The code above creates a SimpleTCP transport implementation with default configuration. This API can be used to create multiple transport instances with the default configuration in a single process by passing unique transport IDs. This API can be used with file-based configurations as long as the matching transport configuration (based upon the transport id) also specifies the same transport type (in our example that is “SimpleTCP”).

We can also configure the transport implementation programmatically, eliminating the need for a configuration file. Here is sample code to create and configure a simple TCP transport implementation.

```
OpenDDS::DCPS::TransportIdType transport_impl_id = 1;
OpenDDS::DCPS::TransportImpl_rch  transport_impl =
  TheTransportFactory->create_transport_impl (
    transport_impl_id, "SimpleTcp", OpenDDS::DCPS::DONT_AUTO_CONFIG);

OpenDDS::DCPS::TransportConfiguration_rch config =
  TheTransportFactory->create_configuration (transport_impl_id);
OpenDDS::DCPS::SimpleTcpConfiguration* transport_config =
  static_cast <OpenDDS::DCPS::SimpleTcpConfiguration*> (config.in());

transport_config->enable_nagle_algorithm_ = true;

if (transport_impl->configure(config.in()) != 0)
  {
std::cerr << "Failed to configure the transport." << std::endl;
return 1;
```

2.1.4.4 Creating a Publisher

Now we are ready to create the publisher and attach the transport implementation we want it to use.

```
DDS::Publisher_var pub =
  participant->create_publisher(
    PUBLISHER_QOS_DEFAULT,
    DDS::PublisherListener::_nil(),
    OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (CORBA::is_nil(pub.in())) {
  std::cerr << "create_publisher failed." << std::endl;
  return 1;
}

// Attach the publisher to the transport.
OpenDDS::DCPS::AttachStatus status = transport_impl->attach(pub.in());
if (status != OpenDDS::DCPS::ATTACH_OK) {
  std::cerr << "Failed to attach to the transport." << std::endl;
  return 1;
}
```

The publisher will now use the pluggable transport instance to which it is attached to publish data samples to the network.

2.1.4.5 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.in (),
                         DATAWRITER_QOS_DEFAULT,
                         DDS::DataWriterListener::_nil(),
                         OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (CORBA::is_nil(writer.in())) {
  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 writer_i =
```

```
        Messenger::MessageDataWriter::_narrow(writer.in());
```

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. Wait on the wait set for a specified period of time
6. Get the publication matched status
7. If the current count of matches is less than one, then go back to step 5 and wait some more
8. If the current count of matches is one or more, detach the condition from the wait set and proceed to publication

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);

DDS::ConditionSeq conditions;
DDS::PublicationMatchedStatus matches = { 0, 0, 0, 0, 0 };
DDS::Duration_t timeout = { 30, 0 };

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

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

} while (matches.current_count < 1);
```

```
ws->detach_condition(condition);
```

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

2.1.4.6 Sample Publication

The message publication is quite straightforward:

```
// Populate instance
Messenger::Message message;
message.subject_id = 99;
message.from       = CORBA::string_dup("Comic Book Guy");
message.subject    = CORBA::string_dup("Review");
message.text       = CORBA::string_dup("Worst. Movie. Ever.");
message.count      = 0;
DDS::ReturnCode_t ret = writer_i->write(message, DDS::HANDLE_NIL);

if (ret != DDS::RETCODE_OK) {
  std::cerr << "MessageDataWriter::write() returned failed, " <<
    "return code = " << ret << std::endl;
  return 1;
}
```

This message is distributed to all connected subscribers that are registered for our topic. 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 2.2.1 for details on using instance handles in publication.

2.1.5 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`.

2.1.5.1 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,
                              DDS::DomainParticipantListener::_nil(),
                              OpenDDS::DCPS::DEFAULT_STATUS_MASK);
    if (CORBA::is_nil (participant.in ())) {
      std::cerr << "create_participant failed." << std::endl;
      return 1 ;
    }
```

2.1.5.2 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.in(), "")) {
      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.in (),
                                TOPIC_QOS_DEFAULT,
                                DDS::TopicListener::_nil(),
                                OpenDDS::DCPS::DEFAULT_STATUS_MASK);
    if (CORBA::is_nil(topic.in())) {
```


Object Computing, Inc.

```
    std::cerr << "Failed to create_topic." << std::endl;
    return 1;
  }
```

2.1.5.3 Initializing and Registering the Transport

We now initialize the Simple TCP transport the same way as in the publisher, using the file-based configuration mechanism.

```
// This value must match the value in the subscriber’s configuration file.
OpenDDS::DCPS::TransportImpl_rch transport_impl =
  TheTransportFactory->create_transport_impl(
    OpenDDS::DCPS::DEFAULT_SIMPLE_TCP_ID,
    OpenDDS::DCPS::AUTO_CONFIG);
```

Next, we create the subscriber with the default QoS and attach the transport, as in the publisher.

```
// Create the subscriber and attach to the corresponding transport.
DDS::Subscriber_var sub =
  participant->create_subscriber(SUBSCRIBER_QOS_DEFAULT,
                                 DDS::SubscriberListener::_nil(),
                                 OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (CORBA::is_nil(sub.in())) {
  std::cerr << "Failed to create_subscriber." << std::endl;
  return 1;
}

// Attach the subscriber to the transport.
OpenDDS::DCPS::AttachStatus status = transport_impl->attach(sub.in());
if (status != OpenDDS::DCPS::ATTACH_OK) {
  std::cerr << "Failed to attach to the transport." << std::endl;
  return 1;
```

2.1.5.4 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.in (),
  DATAREADER_QOS_DEFAULT,
  listener.in(),
  OpenDDS::DCPS::DEFAULT_STATUS_MASK);
if (CORBA::is_nil(dr.in())) {
  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.

2.1.6 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 (CORBA::is_nil(reader_i.in())) {
      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 2.2.

2.1.7 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.in ());
TheTransportFactory->release();
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. Lastly, we release our transport factory and shutdown the service 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. 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;
}
```

2.1.8 Running the Example

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

```
$DDS_ROOT/bin/DCPSInfoRepo -ORBSvcConf tcp.conf -o repo.ior
./publisher -ORBSvcConf tcp.conf
./subscriber -ORBSvcConf tcp.conf
```

The `-ORBSvcConf` configuration directive file dynamically loads and configures the SimpleTCP transport library.

One side effect of using the default QoS properties is that, as we increase the number of samples being published, some of the samples will be dropped as the subscriber falls behind. To avoid dropping samples, we need to either

ensure that the subscriber can keep up or change the QoS settings. QoS policies are described in Chapter 3.

See Chapter 5 for a complete description of the OpenDDS configuration parameters.

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

Object Computing, Inc.

`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 which QoS policies 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 Supported Policies

Listed below are the QoS policies that are currently supported by OpenDDS. Any policy not listed here uses its default value. The default values of unsupported policies are as described in the DDS specification and are discussed in 3.3.

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 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, the QoS changes are propagated to the DCPSInfoRepo via QoS update invocations on the corresponding entity, such as `update_subscription_qos()`. 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, 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 = pub->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
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

OBJECT COMPUTING, INC.

Table 3-2 Default Topic QoS Policies

Policy	Member	Default Value
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

Table 3-5 Default DataWriter QoS Policies

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	BEST_EFFORT_RELIABILITY_QOS[1] 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
USER_DATA	value	(not set)
OWNERSHIP	kind	SHARED_OWNERSHIP_QOS
OWNERSHIP_STRENGTH	value	0
WRITER_DATA_LIFECYCLE	autodispose_unregistered_instances	1

1. 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).

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

Object Computing, Inc.

Table 3-6 Default DataReader QoS Policies

Policy	Member	Default Value
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	DURATION_ZERO_SEC DURATION_ZERO_NSEC

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 periodically polls participants for liveliness. 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
};
```


OBJECT COMPUTING, INC.

```
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 SimpleTCP transport supports the “reliable” value for this policy and the SimpleUDP transport only supports the “best effort” value. 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 3.2.7 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 is “best effort.”

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 liveliness kind of data writer must be greater than or equal to the value of data writer.

Note *Versions of OpenDDS up to 2.0, incorrectly set the default reliability kind of data writers to* `BEST_EFFORT_RELIABILITY_QOS` *instead of* `RELIABLE_RELIABILITY_QOS`.

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;
  Duration_t service_cleanup_delay;
};
```

By default the kind is VOLATILE_DURABILITY_QOS and service_cleanup_delay is zero which means infinite time delay.

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 service_cleanup_delay specifies how long to delay the instance cleanup after the instance is disposed and all data writers unregister the instance.

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 3.2.8 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

OBJECT COMPUTING, INC.

`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 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:

Object Computing, Inc.

- `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. OpenDDS does not currently support group presentation.

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 NO_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.

3.3 Unsupported Policies

The unsupported policies cannot be modified with OpenDDS and always take the default value. The following subsections discuss some of the default values that may affect application behavior.

3.3.1 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).

The Ownership policy is optional for compliance with the Minimum profile and is required for compliance with the Ownership profile. OpenDDS only supports the default ownership kind value of `SHARED`.

3.3.2 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.

The Ownership Strength policy is optional for compliance with the Minimum profile and is required for compliance with the Ownership profile. Setting the Ownership Strength policy has no affect in OpenDDS.

3.4 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.in (),
                         dw_qos,
                         DDS::DataWriterListener::_nil(),
                         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 4.2.

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 4.3.

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 4.4.

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.

OBJECT COMPUTING, INC.

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.

Object Computing, Inc.

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.in (),
      DATAREADER_QOS_DEFAULT,
      listener.in(),
      DDS::DATA_AVAILABLE_STATUS);
```

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

```
dr->set_listener(listener.in(),
                 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

4.4.1 Overview

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.

4.4.2 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.

4.4.3 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`.

4.4.4 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.

4.4.5 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. OpenDDS currently supports only a subset of the query syntax specified by the specification. Only queries of the form "`ORDER BY ...`" are currently supported.

4.4.6 Guard Conditions

The guard condition is a simple interface that allows your application to create its own condition classes and trigger it when certain application events occur.

Object Computing, Inc.

CHAPTER 5

Configuration

5.1 Configuration Files

OpenDDS includes a file-based configuration framework for configuring both global settings as well as transport implementations for publishers and subscribers.This chapter summarizes the configuration settings in OpenDDS.

We use the `-DCPSConfigFile` command-line argument to pass the location of the configuration file into OpenDDS. For example,

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

causes the OpenDDS service participant to read configuration settings from the `pub.ini` configuration file. More accurately, we pass the publisher's command-line arguments to the service participant singleton when we initialize the domain participant factory. We did this 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/DCPS/Service_Participant.h` for details.

5.1.1 Common Configuration Settings

The `[common]` section of the OpenDDS configuration file contains settings for attributes 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=corbaloc:iiop:localhost:12345/DCPSInfoRepo
DCPSLivelinessFactor=80
DCPSChunks=20
DCPSChunksAssociationMultiplier=10
DCPSBitTransportPort=
DCPSBitLookupDurationMsec=2000
DCPSPendingTimeout=30
```

It is not necessary to specify every attribute.

Attribute 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 corbaloc:iiop:localhost:12345/DCPSInfoRepo
```

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

Table 5-1 Common Configuration Settings

Option	Description	Default
`DCPSDebugLevel` *n*	Integer value that controls the amount of debug information the DCPS layer prints. Valid values are 0 through 10.	`0`

Table 5-1 Common Configuration Settings

Option	Description	Default
`DCPSTransportDebugLevel n`	Integer value for controlling the transport logging granularity. Legal values span from 0 to 5.	`0`
`DCPSInfoRepo` *`objref`*	Object reference for locating the DCPS Information Repository	`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`
`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`

Table 5-1 Common Configuration Settings

Option	Description	Default
`DCPSBit [1|0]`	Toggle Built-In-Topic support.	`1`
`DCPSBitTransportPort` *port*	Port used by the Simple TCP transport for Built-In Topics.	none; OS chooses port
`DCPSBitTransportIPAddress`	IP address identifying the local interface to be used by SimpleTcp transport for the Built-In Topics.	empty string; equivalent to `INADDR_ANY`
`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`
`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`
`DCPSPendingTimeout` *sec*	The maximum duration in seconds a data writer will block to allow unsent samples to drain on deletion.	0; blocks indefinitely

Table 5-1 Common Configuration Settings

Option	Description	Default
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; OS chooses time slice.

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).

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.

5.1.2 Transport Configuration Settings

An OpenDDS user may configure one or more transports in a single configuration file. A sample transport configuration is below:

```
[transport_impl_1]
transport_type=SimpleTcp
swap_bytes=0
```

```
optimum_packet_size=8192
```

Again, it is not necessary to specify every attribute.

The “1” in the transport_impl_1 marker is the identifier for the transport. That number must match the transport id passed to `create_transport_impl()` in the code.

```
OpenDDS::DCPS::TransportIdType transport_impl_id = 1;

OpenDDS::DCPS::TransportImpl_rch transport_impl =
  TheTransportFactory->create_transport_impl (transport_impl_id,
                                              OpenDDS::DCPS::AUTO_CONFIG);
```

Thus, we can see where the transport's identifier of “1” in the configuration file maps to the creation of the transport in the code and the configuration settings from that file are applied to that transport implementation.

5.1.2.1 Common Transport Configuration Settings

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

Table 5-2 Transport Configuration Settings

Option	Description	Default
`transport_type` *`transport`*	Type of the transport; the list of available transports can be extended programmatically via the OpenDDS Pluggable Transport Framework. `SimpleTcp`, `SimpleUdp`, `SimpleMcast`, and `ReliableMulticast` are included with OpenDDS.	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 setting between machines. The purpose of this setting is to allow the developer to decide which side will make the endian adjustment, if necessary.	`0`

Table 5-2 Transport Configuration Settings

Option	Description	Default
`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
`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.	0(disabled)
`datalink_release_delay`	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` setting 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.

5.1.2.2 SimpleTcp Transport Configuration Settings

The following table summarizes the transport configuration attributes that are either unique to the Simple TCP transport, or whose default value or description is overridden by the Simple TCP transport:

Table 5-3 SimpleTcp Configuration Settings

Option	Description	Default
`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
`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`
`conn_retry_initial_delay` *n*	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`
`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`
`conn_retry_attempts` *n*	Number of reconnect attempts before giving up and calling the `on_publication_lost()` and `on_subscription_lost()` callbacks.	`3`

Object Computing, Inc.

Table 5-3 SimpleTcp Configuration Settings

Option	Description	Default
`max_output_pause_period` *n*	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_connect_duration` *n*	Timeout (milliseconds) for initial passive connection establishment. This does NOT affect the reconnect timing.	`0 (wait forever)`
`passive_reconnect_duration` *n*	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`

SimpleTcp Reconnection Options

When a TCP connection gets closed DDS 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.

5.1.2.3 SimpleUdp/SimpleMcast Transport Configuration Settings

While both SimpleUdp and SimpleMcast are unreliable datagram transports, they share a set of common transport configuration attributes. The following table summarizes those common transport configuration attributes that are either unique to both SimpleUdp and SimpleMcast transports, or whose

default value or description is overridden by SimpleUdp and SimpleMcast transports:

Table 5-4 SimpleUdp and SimpleMcast Common Configuration Settings

Option	Description	Default
`max_packet_size` *n*	Maximum size of a UDP packet. The `SimpleUdp` and `SimpleMcast` transports have a different default value than the other transports.	`62501`
`max_output_pause_period` *n*	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 socket will be closed and `on_*_lost()` callbacks will be called. If the value is zero, the default, then this check will not be made.	`0`

The SimpleUdp and SimpleMcast share the `local_address` configuration but its meaning is different for the different transport implementations. Here are the settings unique to the SimpleUdp transport:

Table 5-5 SimpleUdp Configuration Settings

Option	Description	Default
`local_address` *host:port*	Address and port at which the transport reads UDP packets. 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

In addition to the common configuration attributes listed above, the SimpleMcast transport specifies a few other configuration attributes. The following table summarizes those configuration attributes that are unique to the SimpleMcast transport.

OBJECT COMPUTING, INC.

Table 5-6 SimpleMcast Configuration Settings

Option	Description	Default
`local_address` *`host:port`*	Used on the publisher side to specify which NIC card will be used. This is not available on the subscriber side; it defaults to use 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
`multicast_group_address` *`host:port`*	Address at which the publisher sends multicast packets to and subscriber receives multicast packets from. Uses ACE default multicast address as default.	224.9.9.2:20001 (IPv4) ff05:0::ff01:1:20001(IPv6)
`receiver` *`0|1`*	Flag indicates if the transport is receiving side (subscriber) or sending side (publisher). Defaults to 0 (publisher) side.	`0`

5.1.2.4 ReliableMcast Transport Configuration Settings

The ReliableMcast transport builds data reliability upon the multicast protocol. There are some similarities between its options and those of the SimpleMcast transport. Here is the full list of ReliableMcast options:

Table 5-7 ReliableMcast Configuration Settings

Option	Description	Default
`local_address` *`host:port`*	Used on the publisher side to specify which NIC will be used. This is not available on the subscriber side; it defaults to use 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
`multicast_group_address` *`host:port`*	Address at which the publisher sends multicast packets to and subscriber receives multicast packets from. Uses ACE default multicast address as default.	224.9.9.2:20001 (IPv4) ff05:0::ff01:1:20001(IPv6)

Table 5-7 ReliableMcast Configuration Settings

Option	Description	Default
receiver *0|1*	Flag indicates if the transport is receiving side (subscriber) or sending side (publisher). Defaults to 0 (publisher) side.	0
sender_history_size n	Specifies the history buffer size for the sender, in units of packets. The history buffer consumes memory but provides recall in the event of dropped packets at any receiver.	1024
receiver_buffer_size n	Specifies the buffer size for a receiver, in units of packets. This buffer lets the receiver properly order incoming packets and detect gaps. A larger buffer will consume memory, while a smaller one would reduce the effectiveness of the reliability protocol.	256

5.1.3 Multiple DCPSInfoRepo Configuration

A single OpenDDS process can be associated with multiple DCPS information repositories (DCPSInfoRepo).

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

Domains not explicitly mapped with a repository are automatically associated with the default repository. Individual DCPSInfoRepos can be associated with multiple domains, however domains cannot be shared between multiple DCPSInfoRepos.

Repository and domain association information is contained within individual [repository] and [domain] subsections within the configuration file. The subsections are specified using a slash separated path syntax. Repository subsection header follow the format [repository/<NAME>] where the "repository/" is literal and "<NAME>" is replaced with an arbitrarily chosen but unique subsection name. Similarly, a domain subsection is specified as [domain/<NAME>]. There may be any number of repository or domain sections within a single configuration file.

OBJECT COMPUTING, INC.

Each repository section requires the keys `RepositoryIor` and `RepositoryKey` to be defined. The `RepositoryKey` values must be unique for each repository within the configuration file.

Each Domain subsection requires the keys `DomainId` and `DomainRepoKey` to be defined. The `DomainRepoKey` matched to a `RepositoryKey` maps the domain to that repository. The special value `DEFAULT_REPO` can be used to associate a domain with the default repository.

Table 5-8 Multiple repository configuration sections

Subsection	Key	Value
`[repository/<NAME>]`	`RepositoryIor`	Repository IOR.
	`RepositoryKey`	Unique key value for the repository.
`[domain/<NAME>]`	`DomainId`	Domain being associated with a repository.
	`DomainRepoKey`	Key value of the mapped repository.

5.2 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.

5.2.1 DCPS Layer Logging

Logging in the DCPS layer of OpenDDS is controlled by the `DCPSDebugLevel` configuration setting 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

5.2.2 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 *Actually, 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.*

Object Computing, Inc.

CHAPTER 6

Pluggable Transports

The Configuration chapter gave an overview of currently available configuration options. What follows is a discussion of the specifics of the individual transports and how their behavior can be modified by using these options.

6.1 Simple TCP Transport

As observed in the previous chapter, there are a number of configurable options for SimpleTCP. 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` parameter is used by the peer to establish a connection. By default, the TCP transport selects a random port number on the NIC with 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 an inter-host test, the `local_address` can not be `localhost` and should be configured with an externally visible interface(i.e. `192.168.0.2`), or you can leave it unspecified in which case the FQDN and a random port will be used. Note that this parameter also applies to unreliable datagram transports with the same restrictions.

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. loopback).

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 `passive_connect_duration` parameter 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. When a FQDN is not found, the system will emit a warning.

SimpleTCP exists as an independent library and therefore needs to be linked and configured like the other pluggable transport libraries. The `-ORBSvcConf` option feeds the ACE Service Configuration directive file to configure the SimpleTCP library. The Messenger example from 2.1.4 demonstrates dynamically loading and configuring the SimpleTCP library.

When the SimpleTCP library is built statically, your application must link directly against the SimpleTCP library. To do this, your application must first include the proper header for service initialization, `$DDS_ROOT/dds/DCPS/transport/simpleTCP/SimpleTcp.h`. Then, the static initialization directive

```
static DCPS_SimpleTcpLoader "-type SimpleTcp"
```

will configure the SimpleTCP transport at run-time.

You can also configure the publisher and subscriber transport implementations programatically, as described in 2.1. Configuring subscribers and publishers should be identical, but different addresses/ports should be assigned to each Transport Implementation.

6.2 Unreliable Datagram Transports

As mentioned in previous sections, two unreliable datagram transports, SimpleUdp and SimpleMcast, are supported in this release. Both transports exist in the `SimpleUnreliableDgram` library. To use these transports, the `SimpleUnreliableDgram` library needs be dynamically or statically linked via the -ORBSvcConf option. You can dynamically load the `SimpleUnreliableDgram` library with a service configuration directive:

```
dynamic OPENDDS_DCPS_SimpleUnreliableDgramLoader Service_Object *
SimpleUnreliableDgram:_make_OPENDDS_DCPS_SimpleUnreliableDgramLoader()
"-type SimpleUdp"
```

With this service configuration directive, the `SimpleUdp` component is registered with the transport factory as the library is loaded. To apply the SimpleMcast transport, replace `SimpleUdp` in the directive above with `SimpleMcast`. A single process can apply both SimpleUdp and SimpleMcast transports via multiple service configuration directives.

Because the unreliable datagram transports do not support fragmentation of a single sample into multiple packets, they currently limit the size of marshaled samples, including headers, to 64 KB. Attempting to send a sample greater than 64 KB with these unreliable datagram transports will result in an error message and the sample not being delivered.

Using the unreliable datagram transport involves the same steps that we have seen before: creating a Transport Implementation, attaching it to the publisher and subscriber objects, and configuring it through one or more configuration files. As observed in the previous section, SimpleUdp and SimpleMcast transport configurations share a common set of attributes. In addition, the SimpleMcast transport has its own specific attributes. The following sections show a transport configuration example for SimpleUdp and SimpleMcast and special notes for the individual attributes.

6.2.1 SimpleUDP Transport

Here is a SimpleUDP transport configuration example:

```
# file pub_udp.ini

[common]
DCPSDebugLevel=0
```

```
DCPSInfoRepo=file://repo.ior

[transport_impl_2]
transport_type=SimpleUdp
local_address=localhost:16701
max_output_pause_period=0
```

According to this configuration file, a publisher application will read UDP packets on port 16701 on the loopback network interface.

Note that the `max_output_pause_period` configuration attribute specifies the timeout when the transport is under backpressure. Unlike SimpleTcp and SimpleMcast transports, backpressure has not been observed during internal testing and development; this parameter and its functionality have been included as such a situation may exist in a DDS deployment environment. Backpressure is handled in a similar manner to SimpleTcp and SimpleMcast.

The example above shows the configuration for a publisher, but a subscriber's configuration may just differ in terms of its `local_address` (IP address and port).

6.2.2 SimpleMcast Transport

Here is a SimpleMcast transport configuration example for a publisher:

```
# file pub_mcast.ini

[common]
DCPSDebugLevel=0
DCPSInfoRepo=file://repo.ior

[transport_impl_3]
transport_type=SimpleMcast
local_address=192.168.0.2:16701
multicast_group_address=224.0.0.1:29803
receiver=0
max_output_pause_period=0
```

In this example, a publisher sends multicast packets to port 29803 on the 224.0.0.1 multicast group address from port 16701 on the NIC with the IP address of 192.168.0.2.

Note that on Win32 machines, the `local_address` parameter should not be the loopback address (`localhost`, or `127.0.0.1`). It must have an external

interface's address or remain blank to let the transport automatically select the default NIC.

Here is the configuration for a SimpleMcast subscriber:

```
# file sub_mcast.ini

[common]
DCPSDebugLevel=0
DCPSInfoRepo=file://repo.ior

[transport_impl_3]
transport_type=SimpleMcast
multicast_group_address=224.0.0.1:29803
receiver=1
max_output_pause_period=0
```

This example configures a subscriber application to listen to the 224.0.0.1 multicast group address, again at port 29803. The same `multicast_group_address` should be used for both publishers and subscribers.

The `receiver` configuration attribute specifies the role of the transport. It must be `0` on the publisher side and `1` on the subscriber side. Unlike SimpleTcp and SimpleUdp transports, the same SimpleMcast transport object cannot be shared by both the publisher and subscriber.

Of particular importance is the missing `local_address` parameter. While it is perfectly acceptable to specify this parameter for a publisher, it is not available for subscribers in this version of OpenDDS.

6.3 Reliable Multicast Transport

The reliable multicast transport provides reliable operation on an unreliable multicast channel. Understanding the meaning of "reliable", how this reliability is achieved, and what happens when reliability is compromised are all vital to properly configuring the transport. Of note to developers is that the reliability components of this transport are completely separate from those that handle sending and receiving of data. Thus, the reliability portion could be extracted and reused in another transport, provided the underlying transport allows for bidirectional communication.

In the context of this transport, reliability is defined as in-order, lossless delivery of data. Since multicast is UDP, it exhibits UDP's loss and transmission characteristics. Therefore, to achieve the desired level of reliability, both the sending and receiving side must have special logic:

A sender must:

- Fragment outgoing messages from the transport framework into packets with headers appropriate for reliability and reassembly.
- Maintain a packet history buffer to respond to retransmission requests.
- Send out periodic heartbeat messages to let receivers detect loss at the end of a burst.
- Respond to requests for expired historical data with a "not available" packet.

A receiver must:

- Buffer data received out of order.
- Detect "gaps" in the transmission and request retransmissions.
- Deliver complete messages to the transport framework in the proper order.
- Report disconnection upon receipt of a "not available" packet for data it has requested and not yet received.

Like other transports ReliableMcast needs to be either dynamically or statically configured. Shown below is directive to dynamically load and configure the transport:

```
dynamic OPENDDS_DCPS_ReliableMulticastLoader Service_Object *
ReliableMulticast:_make_OPENDDS_DCPS_ReliableMulticastLoader() ""
```

Static configuration requires the inclusion of the header file:

```
#include "dds/DCPS/transport/ReliableMulticast/ReliableMulticast.h"
```

and the service config directive:

```
static OPENDDS_DCPS_ReliableMulticastLoader ""
```

CHAPTER 7

Built-In Topics

7.1 Introduction

In OpenDDS, built-in topics (BITs) are published by the `DCPSInfoRepo` server. The InfoRepo's `-NOBITS` command line option may be used to suppress publication of built-in topics. Four separate topics are defined for each domain that a `DCPSInfoRepo` server manages. 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 BITs can be toggled via the `DCPSBit` configuration option (see Table 5-1). 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.

Note *The Built-In Topics feature is currently dependent upon the SimpleTCP transport library. The DCPSInfoRepo server as well as any participating*

subscribers and publishers will need to configure the SimpleTCP library to handle Built-In topics.

Sections 7.3 through 7.6 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.

7.2 Building Without BIT Support

If you are not planning on using Built-in-Topics in your application, you can configure DDS to remove BIT support at build time. Doing so can reduce the footprint of the core DDS library by up to 30%. To remove support for Built-In Topics follow these steps:

1. Regenerate the project files without the Built-In Topic feature. Either use the command line "feature" argument to MPC:

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

   Or alternatively, add the line `built_in_topics=0` to the file `$DDS_ROOT/MPC/config/default.features` and regenerate the project files using MPC.
2. If you are using the gnuace MPC project type (which is the case if you will be using GNU make as your build system), add line "`built_in_topics=0`" to the file `$ACE_ROOT/include/makeinclude/platform_macros.GNU`.
3. Build DDS as usual (see `$DDS_ROOT/docs/INSTALL` for instructions).

7.3 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.

7.4 DCPSTopic Topic

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.

7.5 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.

7.6 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.

7.7 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.in());

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 8

dcps_ts.pl Options

8.1 dcps_ts.pl Command Line Options

The `dcps_ts.pl` script is located in `$DDS_ROOT/bin` and parses a single IDL file for DCPS-enabled types then generates type support code for those types. For each IDL file processed, such as `xyz.idl`, it generates three files: `xyzTypeSupport.idl`, `xyzTypeSupportImpl.h`, and `xyzTypeSupportImpl.cpp`. In the typical usage, the script is passed a number of options and the IDL file name as a parameter. For example,

```
$DDS_ROOT/bin/dcps_ts.pl Foo.idl
```

The following table summarizes the entire set of options the script supports. Note that many have terse and verbose variants of the same option.

Table 8-1 dcps_ts.pl Command Line Options

Option	Description	Default
`--verbose` `--noverbose`	Enables/disables verbose execution	Quiet execution
`--debug` `-d`	Enable debug statements in the script	No debug output

Table 8-1 dcps_ts.pl Command Line Options

Option	Description	Default
`--help` `-h`	Prints a usage message and exits	N/A
`--man`	Prints a man page and exits	N/A
`--dir=`*`dirpath`* `-S `*`dirpath`*	Subdirectory where IDL file is located	No subdir used
`--export=`*`macro`* `-X `*`macro`*	Export macro used for generating C++ implementation code.	No export macro used
`--pch=file`	Pre-compiled header file to include in generated C++ files	No pre-compiled header included
`--module=`*`name`*	Specifies the name of the C++ namespace (and IDL module) into which generated code should be placed.	Generated code is placed in the global namespace.
`--timestamp` `-t`	Backup any previously existing generated files with a timestamp suffix.	Old files are not backed up
`--nobackup`	Do not back up the previously generated files	Old files are not backed up
`--idl=`*`file`*	The IDL file to process.	IDL file is assumed to be a parameter
`--output=outdir` `-o outputdir`	Output directory where dcps_ts.pl should place the generated files.	The current directory
`--extension=`*`module_name`*	Allows additional functionality to be added at run time through another Perl module.	No additional functionality is added at run time.

These options mainly divide into two main categories, those related to the execution of the script and those that control the generated code. In the former category are documentation options like `--help` and `--man` as well as script debugging options like `--verbose` and `--debug`.

The code generation options allow the application developer to use the generated code in a wide variety of environments. The `-dir` option lets you operate on IDL files in other directories and causes the generated IDL code to use the proper paths in the includes. The `--export` option 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 keyword. The `--pch` option is required if the

generated implementation code is to be used in a component that uses precompiled headers. The `--module` option allows you to put the generated C++ code in a namespace (and the generated IDL into a module) to avoid name collisions and pollution of the global name space. The `--timestamp` and `--nobackup` options control whether older versions of the generated files are preserved with timestamp-appended file name or whether they are simply overwritten.

The `--idl` option allows you to specify the IDL file to process with an option instead of with a simple parameter.

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)

Table 9-1 DCPS Information Repository Options

Option	Description	Default
`-FederationId <id>`	Unique identifier for this repository within any federation. This is supplied as a 32 bit decimal numeric value.	N/A
`-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.

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 Simple 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 9.2 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"
```


OBJECT COMPUTING, INC.

will persist InfoRepo 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 InfoRepo can be reincarnated to a prior state.

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`, has a value of 1382379631 (0x5265706f), and should not be used by applications for data distribution.

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 5.1.3.

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 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.

OBJECT COMPUTING, INC.

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 `opendds_repo_ctl` and is located in the `$DDS_ROOT/bin` directory. It has a command format syntax of:

```
opendds_repo_ctl <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 opendds_repo_ctl Repository Management Command

Command	Syntax	Description
join	`opendds_repo_ctl 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	`opendds_repo_ctl 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.

Table 9-3 opendds_repo_ctl Repository Management Command

Command	Syntax	Description
shutdown	opendds_repo_ctl 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	opendds_repo_ctl kill <target>	Kills the <target> repository regardless of its federation status.
help	opendds_repo_ctl 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:

```
opendds_repo_ctl join 2112 otherhost:1812
```

This generates a CORBA object reference of `corbaloc:iiop: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:iiop: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.

Table 9-4 Federation Management Command Arguments

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:iiop: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 5.1.3. 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:iiop: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 2.1.8.

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 2.1.8.

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`	`dcps_ts.pl`: 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}`	`dcps_ts.pl`: DDS type-specific C++ impl.
`Baz/BarTypeSupportImpl.java`	`dcps_ts.pl`: DDS type-specific Java impl.
`Baz/Bar*.java`	`idl2jni`: Java representation of IDL struct
`FooJC.{h,cpp}`	`idl2jni`: JNI native method implementations

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:

   ```
   --- BEGIN Foo.mpc ---
   project: dcps_java {

     idlflags      += -Wb,stub_export_include=Foo_Export.h \
                      -Wb,stub_export_macro=Foo_Export
     dcps_ts_flags += --export=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
     }
   }
   --- END Foo.mpc ---
   ```

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`, 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:

OBJECT COMPUTING, INC.

```
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;
    }
    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;
    }
```

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 Domain Participant 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 = 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.

Object Computing, Inc.

10.4.3 Initializing and Registering the Transport

We now initialize the transport we want to use.

```
TransportImpl transport_impl =
  TheTransportFactory.create_transport_impl(1,
    TheTransportFactory.AUTO_CONFIG);
```

The `TheTransportFactory.AUTO_CONFIG` argument indicates intent to use a configuration file for transport initialization. The supplied transport Id must have a matching entry in the configuration file. The code itself is independent of the transport implementation details.

10.4.4 Creating a Publisher

Next, we create a publisher:

```
Publisher pub = dp.create_publisher(
  PUBLISHER_QOS_DEFAULT.get(),
  null,
  DEFAULT_STATUS_MASK.value);
```

and attach it to the transport we previously initialized:

```
AttachStatus stat = transport_impl.attach_to_publisher(pub);
```

DataWriters and DataReaders spawned from this publisher will use the attached transport.

10.4.5 Creating a DataWriter and Registering an Instance

With the publisher attached to a transport, 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, create the same named topic, and initialize a compatible transport.

```
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);
```

```
TransportImpl transport_impl =
  TheTransportFactory.create_transport_impl(1,
    TheTransportFactory.AUTO_CONFIG);
```

10.5.1 Creating a Subscriber

As with the publisher, we create a subscriber and attach it to the transport:

```
Subscriber sub = dp.create_subscriber(
  SUBSCRIBER_QOS_DEFAULT.get(), null, DEFAULT_STATUS_MASK.value);
AttachStatus stat = transport_impl.attach_to_subscriber(sub);
```

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 DataWriter 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 OpenDDS environment can be cleaned up 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.

```
TheTransportFactory.release();
```

Closes down any open Transports.

```
TheServiceParticipant.shutdown();
```

Shuts down the ServiceParticipant. This cleans up all OpenDDS associated resources.

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 a common property for the DCPSInfoRepo objects location:

```
[common]
DCPSInfoRepo=file://repo.ior
```

and a transport type property:

```
[transport_impl_1]
transport_type=SimpleTcp
```

The `[transport_impl_1]` section contains configuration information for the transport with the id of “1”. This id is used for transport creation in both our publisher and subscriber:

```
TransportImpl transport_impl =
  TheTransportFactory.create_transport_impl(1,
    TheTransportFactory.AUTO_CONFIG);
```

See 1.7 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 -ORBSvcConf tcp.conf -o repo.ior

$JAVA_HOME/bin/java -ea -cp
classes:$DDS_ROOT/lib/i2jrt.jar:$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes
TestPublisher -ORBSvcConf tcp.conf -DCPSConfigFile pub_tcp.ini

$JAVA_HOME/bin/java -ea -cp
classes:$DDS_ROOT/lib/i2jrt.jar:$DDS_ROOT/lib/OpenDDS_DCPS.jar:classes
TestSubscriber -ORBSvcConf tcp.conf -DCPSConfigFile sub_tcp.ini
```

The `-DCPSConfigFile` command-line argument passes the location of the OpenDDS configuration file.

The `-ORBSvcConf` configuration directive file dynamically loads and configures the SimpleTCP transport library.

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.

Index

A

ace x
ADAPTIVE Communication Environment x
Advanced CORBA Programming Using TAO course xvi

B

BIT
See built-in topic
built-in topic 5, 51–52, 97–99, 101, 107–109
DCPSParticipant 5, 98, 101
DCPSPublication 5, 99
DCPSSubscription 5, 100
DCPSTopic 5, 99
example 101

C

c++ example 15–32
C++ Programming with Boost course xvii
common configuration settings 78
common data representation (CDR) 13
common transport configuration 82
compliance x, 7
condition 6
configuration 14–32, 77–89, 128
  common settings 78
  common transport 82
  reliablemcast transport 87
  simplemcast transport 85
  simpletcp transport 84
  simpleudp transport 85
  transport 81
configuring multiple DCPSInfoRepos 88
CORBA Programming with C++ course xvi
create_participant() operation 20, 122
create_topic() operation 26
customer support
  See support

D

data distribution service (DDS)
  data local reconstruction layer (DLRL) 1
  data-centric publish-subscribe (DCPS) 1
    overview 2–6
  platform independent model (PIM) 1
  platform specific model (PSM) 1
data local reconstruction layer (DLRL) 1
data marshaling 4–5, 13, 17
data reader 5–6, 14, 17, 27–30, 33, 43–46, 48–52, 60, 62, 97, 100–101
data sample 4–5, 14, 16–17, 25, 28–33, 45–46, 48, 62, 93, 101
data writer 4–5, 17, 23, 25, 33, 43–52, 60, 97, 99–100, 123
data-centric publish-subscribe (DCPS) 1
  information repository 13, 19, 31, 79, 97, 107–109

overview 2–6

DataReaderListener implementation 28

DataReaderListener interface 28

DCPS

See data-centric publish-subscribe (DCPS)

DCPS_DATA_KEY pragma 16

DCPS_DATA_TYPE pragma 16

DCPS_TRANS_VERBOSE_DEBUG environment variable 108

dcps_ts.pl program 15, 17–18, 103

DCPSBit option 80

DCPSBitLookupDurationMsec option 80

DCPSBitTransportIPAddress option 80

DCPSBitTransportPort option 80

DCPSChunkAssociationMultiplier option 79

DCPSChunks option 79, 81

DCPSConfigFile option 77

DCPSDebugLevel option 78

DCPSInfoRepo 13, 19, 31, 79, 97, 107–109

configuration 88

configuring multiple 88

federation 108–115

example 113

management 111

DCPSInfoRepo option 79

DCPSLivelinessFactor option 79

DCPSParticipant built-in topic 5, 98, 101

DCPSPendingTimeout option 80

DCPSPersistentDataDir option 80

DCPSPublication built-in topic 5, 99

DCPSSubscription built-in topic 5, 100

DCPSTopic built-in topic 5, 99

DCPSTransportDebugLevel option 79

DDS_ROOT environment variable 11–12, 15, 17, 19, 25, 103

deadline policy 49, 60

delete_contained_entities() operation 31

dispose() operation 30

DLRL

See data local reconstruction layer (DLRL)

domain 4–5, 17, 26, 49, 97–100, 110

domain participant 5, 31, 51, 56, 97–101

domain participant factory 20, 31, 56, 77

durability policy 46

durability_service policy 47

E

enable() operation 56
entity_factory policy 55
environment variables
DCPS_TRANS_VERBOSE_DEBUG 108
DDS_ROOT 11–12, 15, 17, 19, 25, 103
example
built-in topic 101
c++ 15–32
DCPSInfoRepo federation 113
java 121–129
quality of service (QoS) policy 61

F

factory
domain participant 20, 31, 56, 77
transport 12, 31
find_topic() operation 26
frequently asked questions (FAQ) xiii

G

group_data policy 51

H

history policy 45, 62

I

IDL compiler
Gdcps option 13, 16, 18
instance xii, 4, 13, 16, 25, 32–33, 45–46, 48, 60, 62, 97, 99–100
interface
DataReaderListener 28
type-specific 17, 29
interface definition language (IDL)
tao_idl program 16
Introduction to CORBA course xv

J

java bindings 117–129
jms support 129
java example 121–129
java message service (jms) 129

L

latency_budget policy 53
licensing terms ix
lifespan policy 50
listener 6, 14, 20, 23, 27–29, 44, 49–50, 125
liveliness policy 43, 60, 109

M

Make Project Creator x
marshaling 4–5, 13, 17
mpc x
multithreading 14

O

Object Computing, Inc. (OCI) ix–x, xiv–xv, 1
Object Management Group (OMG) specifications
Data Distribution Service for Real-Time Systems (formal/07-01-01) ix–x, 1–2
Object Oriented Design Patterns and Frameworks course xvi
on_data_available() operation 29
on_liveliness_changed() operation 44
on_offered_deadline_missed() operation 49
on_requested_deadline_missed() operation 50
open source software ix
OpenDDS Programming with C++ course xvii
opendds_repo_ctl program 111
operations
create_participant() 20, 122
create_topic() 26
delete_contained_entities() 31
dispose() 30
enable() 56
find_topic() 26
on_data_available() 29
on_liveliness_changed() 44
on_offered_deadline_missed() 49
on_requested_deadline_missed() 50
ORB_init() 20
register_instance() 25, 32
register_type() 20, 122
return_loan() 34
set_qos() 38
string_to_object() 81
take_instance() 33
take_next_instance() 33
take_next_sample() 33
take() 33
unregister() 30
update_subscription_qos() 38
wait_for_acknowledgements() 31
options
DCPSBit 80
DCPSBitLookupDurationMsec 80
DCPSBitTransportIPAddress 80

OBJECT COMPUTING, INC.

DCPSBitTransportPort 80
DCPSChunkAssociationMultiplier 79
DCPSChunks 79, 81
DCPSConfigFile 77
DCPSDebugLevel 78
DCPSInfoRepo 79
DCPSLivelinessFactor 79
DCPSPendingTimeout 80
DCPSPersistentDataDir 80
DCPSTransportDebugLevel 79
ORBSvcConf 31
scheduler 81
scheduler_slice 81
ORB_init() operation 20
ORBSvcConf option 31
ownership policy 60–61
ownership_strength policy 60–61

P

partition policy 49, 52
platform independent model (PIM) 1
platform specific model (PSM) 1
platform support xiv
pluggable transports 12–13, 23–96
policy
quality of service (QoS) 5–6, 14, 20–21, 23, 37–62
deadline 49, 60
durability 46
durability_service 47
entity_factory 55
example 61
group_data 51
history 45, 62
latency_budget 53
lifespan 50
liveliness 43, 60, 109
ownership 60–61
ownership_strength 60–61
partition 49, 52

reader_data_lifecycle 59
reliability 44, 62
resource_limits 48, 62
supported 37
time_based_filter 59
topic_data 51
transport_priority 52
unsupported 60
user_data 51
writer_data_lifecycle 58
policy example 61
pragma
DCPS_DATA_KEY 16
DCPS_DATA_TYPE 16
publisher 4–6, 12–15, 17–19, 23, 25, 27, 30–31, 38, 45, 49–52, 56, 61–62, 93, 123

Q

quality of service (QoS) policy 5–6, 14, 20–21, 23, 37–62
deadline 49, 60
durability 46
durability_service 47
entity_factory 55
group_data 51
history 45, 62
latency_budget 53
lifespan 50
liveliness 43, 60, 109
ownership 60–61
ownership_strength 60–61
partition 49, 52
reader_data_lifecycle 59
reliability 44, 62
resource_limits 48, 62
supported 37
time_based_filter 59
topic_data 51
transport_priority 52
unsupported 60
user_data 51

writer_data_lifecycle 58
quality of service (QoS) policy example 61

R

reader_data_lifecycle 59
register_instance() operation 25, 32
register_type() operation 20, 122
reliability policy 44, 62
reliablemcast transport 95
configuring 87
resource_limits policy 48, 62
return_loan() operation 34

S

sample 4–5, 14, 16–17, 25, 28–33, 45–46, 48, 60, 62, 93, 101
scheduler option 81
scheduler_slice option 81
set_qos() operation 38
simplemcast transport 94
configuring 85
simpletcp transport 12, 15, 21, 27, 45, 108
configuring 84
simpleudp transport 45, 93
configuring 85
single-copy read 33
specification
compliance x, 7
string_to_object() operation 81
subscriber 5–6, 12–15, 17–18, 24–27, 30–32, 45–46, 49–52, 56, 93, 97, 101, 124–125, 127
support xiv
supported platforms xiv
supported policies 37

T

take_instance() operation 33
take_next_instance() operation 33
take_next_sample() operation 33
take() operation 33
tao x, xii
The ACE ORB x, xii
time_based_filter 59
topic 4–5, 14, 16, 20–21, 23, 25–26, 28, 31, 43–51, 97–101, 108
topic_data policy 51
training xv, xvii
- Advanced CORBA Programming Using TAO xvi
- C++ Programming with Boost xvii
- CORBA Programming with C++ xvi
- Introduction to CORBA xv
- Object Oriented Design Patterns and Frameworks xvi
- OpenDDS Programming with C++ xvii
- Using the ACE C++ Framework xvi

transport 12, 14–15, 21, 27, 45, 93, 107–108
- pluggable 12–13, 23–96
- reliablemcast 95
- simplemcast 94
- simpletcp 12, 15, 21, 27, 45, 108
- simpleudp 45, 93

transport configuration 81
transport factory 12, 31
transport_priority policy 52
type-specific interface 17, 29
type-support code generation
- dcps_ts.pl program 15, 17–18, 103

U

unregister() operation 30
unsupported policies 60
update_subscription_qos() operation 38
user_data policy 51
Using the ACE C++ Framework course xvi

W

wait set 6
wait_for_acknowledgements() operation 31
writer_data_lifecycle 58

Z

zero-copy read 33