OpenSplice DDS

Version 3.4

C Tutorial Guide

PRISMTECH

OpenSplice DDS

C Tutorial Guide

Part Number: OS-CTG

Doc Issue 17, 27 May 2008

Copyright Notice

Contents

Table of Contents

Preface

About the C Tutorial Guide

The *C Tutorial Guide* introduces OpenSplice's main concepts, aided by code examples which use the OpenSplice API to create a *chat room* using OpenSplice's publish and subscribe features in order to enable users to efficiently communicate with each other.

The tutorial examples progress from introducing basic concepts, gradually developing them through to a complete application. The complete source code for the example are listed in the *Appendices*[1].

i Please note that the *C Tutorial Guide* is not intended to covers all aspects of OpenSplice, but simply to introduce essential concepts and enable users to begin using OpenSplice as quickly as possible.

The OpenSplice DDS API is embedded in different programming languages. The *C Tutorial Guide* covers the C version of OpenSplice: refer to the appropriate tutorial version for the other supported languages. Example code for all supported languages are listed in the *C Tutorial Guide's Appendices*.

Intended Audience

The *C Tutorial Guide* is intended to be used by C programmers who are using OpenSplice to develop applications.

Organisation

Chapter 1, *Introduction to OpenSplice DDS*, provides an introduction about OpenSplice DDS product and the OMG DDS standard which OpenSplice DDS is based on. This chapter explains the various DDS profiles and the extent that OpenSplice supports them. Also, the tools which are included with OpenSplice are briefly described. *Introduction to OpenSplice DDS* can be skipped if you are already familiar with OpenSplice.

Chapter 2, *A DDS-based Chatroom*, describes the high-level architecture of an example chatroom application, called *Chat*, which the *C Tutorial Guide* uses to explain how to develop applications using OpenSplice. The chapter also analyses the example application is constructed from autonomous components.

Chapter 3, *Data Modelling*, explains how to define data models in IDL and how to translate this IDL model into your chosen language, including how to represent the IDL in the C language.

---

1. Please note that the examples provided in this guide are intended for instructional purposes only and have not been optimised for resource usage.

Chapter 4, *Managing Domains and Topics* describes the initial steps that are needed to connect an application to a DDS Domain as well as how to define the topics the application will use in the Domain. This chapter explains concepts and skills that are needed for subsequent steps in developing an application, such as creating and deleting Entities by means of a factory, error handling and tailoring QoS settings.

Chapter 5, *Publishing the Data*, and Chapter 6, *Subscribing to Data*, describes how to publish data and make subscriptions for accessing information, respectively. A primitive version of a message board, called *MessageBoard*, that sends all incoming chat messages to your screen is introduced.

Chapter 7, *Content-Subscription Profile and Listeners* further develops the message board application by adding *content awareness* through the use of filters, queries and event-based data notification.

Chapter 8, *Waiting for Conditions*, describes how to display user activity and how to keep track of usage history in the chat room through the use of *Conditions*, *WaitSets*, and Quality of Service policies (QosPolicy) which are employed in a *UserLoad* application.

The *Appendices* contain listings of all example source code used in the *C Tutorial Guide*, plus the code listings for the other languages supported by *OpenSplice*.

The *Bibliography* contains a list of references used by the guide and which also may provide useful or essential information.

Conventions

The conventions listed below are used to guide and assist the reader in understanding the C Tutorial Guide.


Item of special significance or where caution needs to be taken.

i
Item contains helpful hint or special information.

WIN
Information applies to Windows (e.g. NT, 2000, XP) only.

UNIX
Information applies to Unix based systems (e.g. Solaris) only.

C
C language specific

C++
C++ language specific

Java
Java language specific

Hypertext links are shown as *blue italic underlined.*

On-Line (PDF) versions of this document: Items shown as cross references, e.g. *Contacts* on page xi, are as hypertext links: click on the reference to go to the item.

```
%  Commands or input which the user enters on the
   command line of their computer terminal
```

`Courier` fonts indicate programming code and file names.

Extended code fragments are shown in shaded boxes:

```
NameComponent newName[] = new NameComponent[1];

// set id field to “example” and kind field to an empty string
newName[0] = new NameComponent (“example”, “”);
```

Italics and ***Italic Bold*** are used to indicate new terms, or emphasise an item.

Arial Bold is used to indicate user related actions, e.g. **File | Save** from a menu.

Step 1: One of several steps required to complete a task.

Contacts

PrismTech can be reached at the following contact points for information and technical support.

Corporate Headquarters

PrismTech Corporation
6 Lincoln Knoll Lane
Suite 100
Burlington, MA
01803
USA

Tel: +1 781 270 1177
Fax: +1 781 238 1700

European Head Office

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK

Tel: +44 (0)191 497 9900
Fax: +44 (0)191 497 9901

Web: *http://www.prismtech.com*
General Enquiries: *info@prismtech.com*

OpenSplice DDS C Tutorial

CHAPTER

1 Introduction to OpenSplice DDS

This section starts by introducing the concepts and philosophies behind the Object Management Groups Data Distribution System (OMG DDS) standardization process. It will explain the characteristics of the different DDS profiles, and will explain how these profiles are incorporated in the OpenSplice DDS product. Then it will provide a short impression of the basic architecture of OpenSplice DDS and how this influences issues like scalability and configuration, followed by a detailed overview of all the benefits that the OpenSplice DDS product will offer you. Finally the OpenSplice DDS Productivity Tools are introduced and it is explained how these might dramatically decrease the costs of your development and maintenance efforts.

1.1 Overview

Real-time availability of information is of utmost importance in the large class of network-centric systems. Information generated from multiple sources must be distributed and made available to 'interested parties' taking into account Quality of Service (QoS) offerings by information-producers and requests by information-consumers. Especially in real-time and mission-critical systems, getting 'the right data at the right time at the right place' is not a trivial task at all and up until recently, there were no standards nor COTS products that addressed this challenge in an integrated solution. The OMG recognized this need for a Data Distribution Service (DDS) and organized members with vast experience in both the 'underlying' technologies (networking and information-management) as well as 'user-level' requirements (distributed, real-time and mission-critical system characteristics), including Thales Naval Netherlands, to join forces and these members defined the 'OMG-DDS' service. The OMG-DDS service specifies a coherent set of profiles that target real-time information-availability for domains ranging from small-scale embedded control systems up to large-scale enterprise information management systems. Each DDS-profile adds distinct capabilities that define the service-levels offered by DDS in order to realize this '*right data at the right time at the right place*' paradigm:

- ***Minimum Profile*** - this *basic* profile utilizes the well known publish/subscribe paradigm to implement highly efficient information dissemination between multiple publishers and subscribers that share interest in so called 'topics'. Topics are the basic data structures expressed in the OMG's IDL language (allowing for automatic generation of typed 'Readers' and 'Writers' of those 'topics' for any mix of languages desired). This profile also includes the QoS framework that allows

the middleware to 'match' requested and offered Quality of Service parameters (the minimum profile offering basic QoS attributes such as 'reliability', 'ordering' or 'urgency').

- ***Ownership Profile*** - this 'replication' profile offers support for replicated publishers of the same information by allowing a '*strength*' to be expressed by each publisher so that only the 'highest strength' information will be made available to interested parties.
- ***Content Subscription Profile*** - this 'content awareness' profile offers powerful features to express fine grained interest in specific information content (content filters). This profile also allows applications to specify *projection views* and *aggregation* of data as well as dynamic *queries* for subscribed 'topics' by utilizing a subset of the well known SQL language whilst preserving the real-time requirements for the information access.
- ***Persistence Profile*** - this 'durability' profile offers transparent and fault tolerant availability of 'non volatile' data that may either represent persistent 'settings' (to be stored on mass media throughout the distributed system) or 'state' preserved in a fault tolerant manner outside the scope of transient publishers (allowing late joining applications and dynamic reallocation).
- ***DLRL Profile*** - this 'object model' (Data Local Reconstruction Layer) extends the previous four data centric *DCPS* profiles with an *object-oriented view* on a set of related topics thus providing typical OO features such as navigation, inheritance and use of value types.

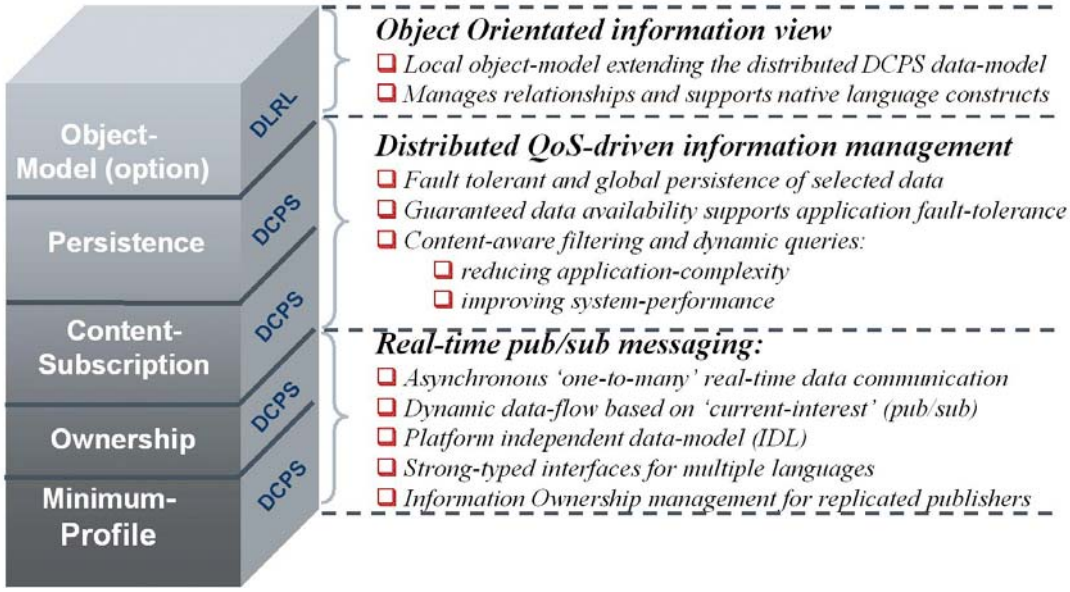

Figure 1 OMG DDS Layers

1.2 OpenSplice DDS Summary

PrismTech's OpenSplice DDS, is a second generation, fully compliant OMG DDS implementation, offering support for all the DCPS profiles (minimum profile, ownership profile, content subscription profile and persistence profile) as well as the DLRL object profile. OpenSplice DDS was initially developed as SPLICE-DDS by Thales Naval Netherlands (TNL), one of the co-authors of the DDS specification and is the result of TNL's over 15 year experience in developing distributed information systems for naval Combat Management Systems (CMS). This field proven middleware is used as the 'information backbone' of TNL's TACTICOS CMS currently deployed in 15 navies around the world. OpenSplice DDS is the 2nd generation COTS evolution of this successful product and consists of several modules that cover the full OMG specification as well as provision of total lifecycle support by an integrated productivity tool suite:

- **OpenSplice DDS core modules** cover the "Minimum" and "Ownership" profiles that provide the basic publish-subscribe messaging functions. The minimum profile is meant to address real time messaging requirements, where performance and low footprint are essential. The ownership profile provides basic support for replicated publishers where 'ownership' of published data is governed by 'strength' indicating the quality of published information.

- **OpenSplice DDS *content subscription* and *persistence* profiles** provide the additional information management features, key for assuring high information availability (fault tolerant persistence of non-volatile information) as well as powerful 'content aware' features (filters and queries), thus enabling unmatched performance for the full range of small scale embedded up to large scale fault tolerant systems.

Free evaluation licenses of OpenSplice DDS are available by e-mailing *sales@prismtech.com*. Currently supported platforms include Solaris Sparc, Linux x86, x86 and VxWorks PowerPC, whereas supported languages are C, C++ (standalone or in seamless cohabitation with any ORB and related C++ compiler) and Java.

1.3 OpenSplice DDS Architecture

1.3.1 Overall

To ensure scalability, flexibility and extensibility, OpenSplice DDS has an internal architecture that utilizes shared memory to 'interconnect' not only all applications that reside within one computing node, but also 'hosts' a configurable and extensible set of services. These services provide 'pluggable' functionality such as networking (providing QoS driven real-time networking based on multiple reliable multicast 'channels'), durability (providing fault tolerant storage for both real-time 'state' data as well as persistent 'settings'), and remote control & monitoring 'soap service' (providing remote web based access using the SOAP protocol from the OpenSplice DDS Tuner tools).

1.3.2 Scalability

OpenSplice DDS utilizes a shared-memory architecture where data is physically present only once on any machine, and where smart ad-ministration still provides each subscriber with his own private 'view' on this data. This allows a subscriber's data cache to be perceived as an individual 'database' that can be content-filtered, queried, etc. (using the content-subscription profile as supported by OpenSplice DDS). This shared-memory architecture results in an extremely low foot-print, excellent scalability and optimal performance when compared to implementations where each reader/writer are 'communication-endpoints' each with its own storage (in other words, historical data both at reader and writer) and where the data itself still has to be moved, even within the same physical node.

1.3.3 Configuration

The OpenSplice DDS middleware can be easily configured 'on the fly' by specifying (only the needed) services to be used as well as configuring those service for optimal matching with the application domain (networking parameters, durability levels, etc). Easily maintainable XML files are utilized to configure all OpenSplice

services. OpenSplice DDS configuration is also supported by means of the MDA tool set allowing system/network modelling and automatic generation of the appropriate XML configuration files.

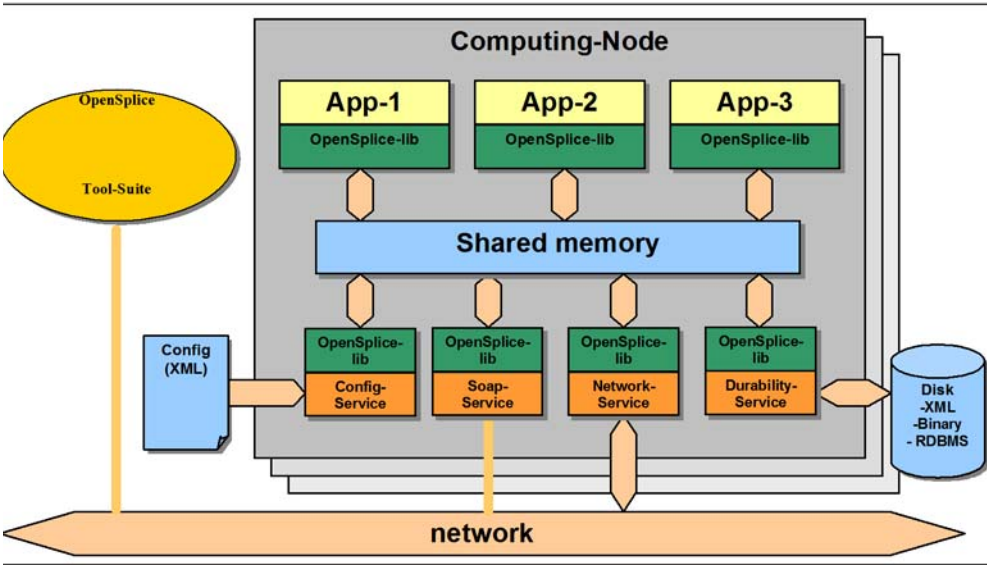

Figure 2 OpenSplice DDS Pluggable Service Architecture

i *Figure 2* only shows one node whereas there are typically many nodes within a system.

1.4 OpenSplice DDS Implementation Benefits

Table 1 below shows the following aspects of OpenSplice DDS, where:

Features significant characteristics of OpenSplice
Advantages shows why a feature is important
Benefits describes how users of OpenSplice can exploit the advantages

Table 1 OpenSplice DDS Features and Benefits

	Features	Advantages	Benefits
General	Information-centric	Enable dynamic, loosely coupled system.	Simplified & better scalable architectures
	Open standard	'Off the shelf' solutions	Lower cost, no vendor lock in
	Built on proven technology	Intended for most the demanding situations.	Assured quality and applicability
	TNN/PT 'inheritance'	Decade long of 'DDS' experience	Proven suitability in mission critical domain
Functional	Real-time pub/sub	Dynamic/asynchronous data communication	Autonomous decoupled applications
	Persistence profile	Fault tolerant data persistence	Application fault tolerance and data high availability
	Content-sub. Profile	Reduced complexity & higher performance.	Easier application design & scalable systems
Performance	Shared memory	low footprint, instant data availability	Processor Scalability
	Smart networking	Efficient data transport	Network Scalability
	Extensive IDL sup.	Includes unbounded strings, sequences	Data Scalability
Usability	Multiple language	Any (mix) of C, C++, Java, Ada	Supports (legacy) code, allows hybrid systems
	Multiple platforms	Any (mix) of Enterprise & RTE Oss	Intercons, enterprise and embedded systems
Tooling and Ease of use	All metadata at runtime	Dynamic discovery of all 'entity info'	Guaranteed data integrity
	Powerful tooling	Support for complete system lifecycle	Enhanced productivity and System Integration
	Remote connect	Web based remote access & control	Remote diagnostics using standard protocols
Legend:	**Equal** to competition	**Better** than competition	**Far surpassing** competition

1.4.1 OpenSplice DDS Tuner

The 100% Java based OpenSplice DDS Tuner tool greatly aids the design, implementation, test and maintenance of OpenSplice-based distributed systems:

- ***Design*** - During the design phase, once the information model is established (in other words, topics are defined and 'registered' in a runtime environment, which can be both a host environment as well as a target environment), the OpenSplice DDS Tuner allows creation of publishers/writers and subscribers/readers on the fly to experiment and validate how this data should be treated by the middleware regarding persistence, durability, latency, etc.
- ***Implementation*** - During the implementation phase, where actual application level processing and distribution of this information is developed, the OpenSplice DDS Tuner allows injection of test input data by creating publishers and writers 'on the fly' as well as validating the responses by creating subscribers and readers for any produced topics.
- ***Test*** - During the test phase, the total system can be monitored by inspection of data (by making 'snapshots' of writer and reader history caches) and behaviour of readers & writers (statistics, like how long data has resided in the reader's cache before it was read).
- ***Maintenance*** - Maximum flexibility for planned and 'ad hoc' maintenance is offered by allowing the 100% JAVA based OpenSplice DDS Tuner tool suite (which can be executed on any JAVA enabled platform without the need of OpenSplice DDS to be installed) to remotely connect via the web based SOAP protocol to any 'reachable' OpenSplice DDS system around the world (as long a HTTP connection can be established with the OpenSplice DDS computing nodes of that system). Using such a dynamic connection, critical data may be logged and data sets may be 'injected' into the system to be maintained (such as new settings which can be automatically 'persisted' using the QoS features as offered by the 'persistence profile supported by OpenSplice DDS).

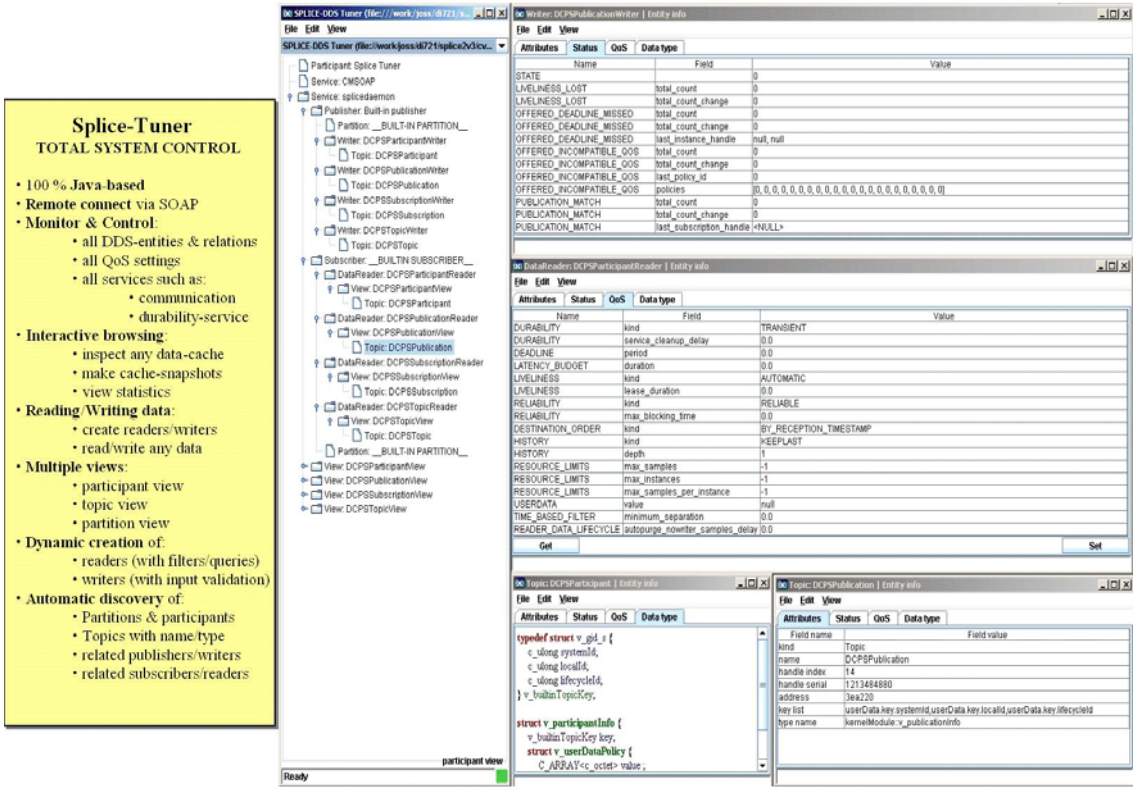

Figure 3 OpenSplice DDS Tuner

1.5 Conclusion

PrismTech's OpenSplice DDS product complemented by its tool sup-port together encompass the industry's most profound expertise on the OMG's DDS standard and products.

The result is unrivalled functional DDS-coverage and performance in large-scale mission-systems, fault-tolerance in information availability, and total lifecycle support including round-trip engineering. A complete DDS solution to ensure a customer's successful adoption of this exciting new technology and to support delivery of the highest-quality applications with shortest time to market in the demanding real-time world.

CHAPTER

2 A DDS-based Chatroom

This section introduces the basic architecture of a Chatroom that is based on OpenSplice DDS. Each subsequent section will elaborate on this basic architecture: a data model will be defined first, then the publishing side will be created, followed by the subscribing side, which will be developed in a number of iterations, increasing its functionality step by step. Finally a monitor will be added that keeps track of the number of Chatters that are currently logged on to the Chatroom.

2.1 Client-Server vs Peer-to-Peer

In this tutorial we want to build an application that uses OpenSplice DDS to distribute chat messages. Traditionally, chatrooms are examples of common client-server architectures, where clients (the chatters) connect to a server (the chatroom) and identify themselves by giving their user name. (In most cases they will have to confirm their identity by providing a password as well.). After the server has recorded their identity, the clients can send as many chat messages as they like. The chatroom collects the chat messages of each client and will forward them to all other participating clients. New clients can request to join a chatroom at any moment in time: they will then have to identify themselves to the server, and the server will make sure that all chat messages received from that moment on will also be forwarded to the newly added client. An example of such a typical client-server approach is presented in *Figure 4*.

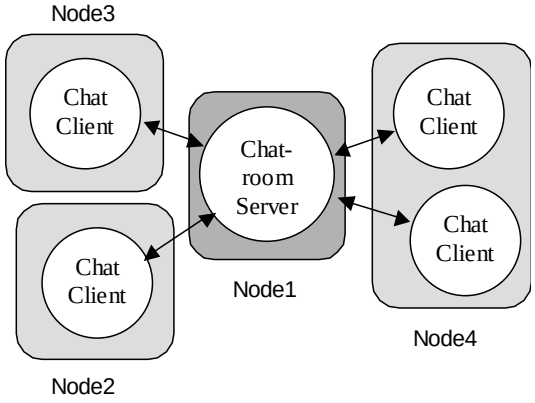

Figure 4 Client-Server Based Approach for a Chatroom

As can be seen from this example, the server is the single point of failure. If it fails, all chatter applications get disconnected. On top of that, every connection is point-to-point, meaning that every chat message is forwarded to each client individually. If the number of connected clients is doubled, the number of messages transmitted from the server is doubled as well. (Provided that the newly added clients do not transmit any chat messages of their own, which would increase the network load even further and could even quadruple it.)

To provide for a more efficient chatter approach, we will employ the DDS-DCPS. The idea is to remove the Chatroom server altogether and let the chat applications (which can now no longer be called clients) directly communicate with each other. The architecture will then become less centralized and will look more like the picture presented in *Figure 5*.

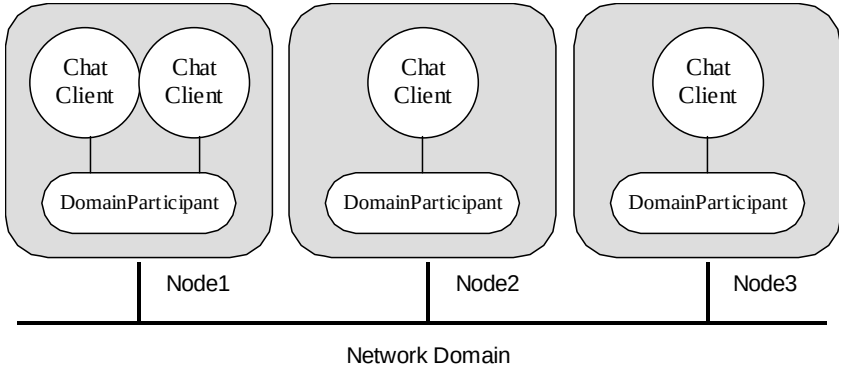

Figure 5 DDS-based Approach

As can be seen from this picture, all applications are equal; there is no centralized point of failure. If a node crashes, all Chatters on that node die, but all the others can keep communicating with each other. What's more, every chat message only has to be transmitted over the network once (using either multicast or broadcast) to deliver it to all the other interested Chatters. Scaling up the number of Chatter applications does not use up any more bandwidth, except of course for the messages sent by these newly added Chatters.

2.2 Analysing the Chatroom Example

In order to focus on the DDS aspect of our Chatroom example, and not on things such as its graphical representation, we will have to break down the problem into several autonomous applications. The following separate applications are distinguished:

- ***Chatter*** - This part is responsible for publishing the identity of the user, followed by all chat messages he or she wishes to transmit. (This application is write only.)
- ***MessageBoard*** - This part is responsible for subscribing itself to all chat messages and for displaying them in the order in which they are received. (This application is read-only).
- ***UserLoad*** - This part is responsible for continuously keeping track of users that join and leave the Chatroom. (This application is read only).

Each of these functional parts will be modelled as a separate process, each one using the standard output to print its messages. Although this constitutes a very primitive User Interface, it completely separates user input from user output thus completely removing the need for any layout related function calls. This helps us to focus our applications almost entirely on efficient utilization of the DCPS, which is the main purpose of this tutorial.

CHAPTER

3 Data Modelling

OpenSplice DDS distributes its data in structured data types, which are transported by means of topics. The first step when using OpenSplice DDS consists of defining these data types. Since OpenSplice can be used on several different platforms with several different programming languages, OMG IDL is used as a language and platform independent modelling language.

This section starts by introducing some basic DDS terminology, which is required to understand the conceptual differences between topics, data types, samples and instances. After that, it will explain which subset of IDL you may use to model your data types, and how to annotate this model with your key field definitions. Finally it will explain how to use the OpenSplice preprocessor to compile the IDL model into your language of choice.

3.1 Data Types, Samples and Instances

All data you want to distribute using OpenSplice DDS has to be defined as a topic. A topic is an aggregation of a structured data type, a keylist, and a specific Quality of Service (QoS) annotation. The keylist is specified as part of the data-type, and identifies the keyfields for that data type. These keyfields can be used to uniquely identify instances of the data type in question, which is a very common approach in relational modelling.

A topic is identified by a topic name that is unique in the context of the Domain where it is used. Note that a topic name and a type name represent two different things: the type name represents the name of the structured data type, the topic name represents the aggregation of this data type with a specific QoS annotation. One data type can be used in several different topic definitions (using different or even the same QoS annotations).

To clarify the efficient usage of topics and to avoid confusion, some basic DDS terms will have to be defined in more detail first:

- ***Data type*** - A DCPS data type represents the definition of a piece of information and is normally declared in IDL as a structured datatype. A data type may embed any number of other data types, but cyclic nesting data types is not possible. Datatypes that are to be distrubuted using topics must be annotated by a declaration of the key fields for that data type.
- ***Sample*** - A DCPS sample represents an allocated data type: in other words, a set of attribute values that is to be distrubuted using a topic.

- ***Keyfield*** - Some fields of a structured datatype can be annotated as being keyfields. The combined values of all keyfields in a sample make up the identity of the item whose state the sample describes.
- ***Instance*** - A DCPS instance represents the notion of a specific observable item, whose state at a certain moment in time can be represented by a sample of a specific data type. The observable item is uniquely identified by the values of its key fields: two samples with different key values represent the states of different instances; two samples with the same key values represent the state of the same instance (but probably these samples represent the state of the instance at different moments in time).

3.2 Modelling Data Types in IDL

A data type represents a structured data type, like an IDL `struct` with several members and a keylist. Whenever you want to read or write topics, you will actually be reading or writing samples of a specific data type. The definition of each data type you will be using has to be written in (a subset of) OMG IDL. The keylist cannot be expressed in IDL, so OpenSplice DDS introduced a special `#pragma` statement for that purpose. [1]

For our chatter application, we will have to define the data types that need to be used to exchange messages between several chatters. We will need at least one topic to transmit the chat messages, and these messages must be accompanied by the user ID of its sender. We can of course use the sender's username as the user ID, but this will mean that the topic's key field will be represented by a string, which may be expensive to process. For this reason, and also for some illustrational purposes, we will decide to make the user ID a 32 bit integer (in other words, an IDL `long`), and to introduce a second topic that maps this user ID to the user's name.

When a Chatter application starts, it will make its existence known to the world by publishing a `NameService` instance, containing a unique `userID` value and the name of the user (which can not be longer than 32 bytes, excluding the '`\0`' terminator according to the IDL). The `userID` field will act as a key to find the corresponding username. After the application has published his `userID` and username, it can start sending chat messages into the world. Each chat message is represented by a `ChatMessage` instance, containing the `userID` of its sender (which acts as its key field), a sequence number expressing the number of chat messages already transmitted, and the message itself, which is an unbounded `string`. Unbounded strings can be of arbitrary length. The resulting topic model is presented below:

1. The use of customized pragma statements is compliant with the IDL standard.

```
1    module Chat {
2      const long MAX_NAME = 32;
3      typedef string<MAX_NAME> nameType;
4
5      struct ChatMessage {
6        long      userID;          // owner of message
7        long      index;           // message number
8        string    content;         // message body
9      };
10   #pragma keylist ChatMessage userID
11
12     struct NameService {
13       long     userID;           // unique user identification
14       nameType name;             // name of the user
15     };
16   #pragma keylist NameService userID
17   };
```

In line *1* a `module` called *`Chat`* is opened, that acts as a scope for all the following declarations. Line *5* introduces the structured data type called *`ChatMessage`*, that contains all the information that is required to identify a specific chat message. Line *10* defines the keylist for this data type (using the `#pragma keylist` statement): it first identifies the data type to which it applies by name, followed by a list of the names of all attributes that represent its key fields (use spaces in case of multiple key fields).

Although the definition of `ChatMessage` is fully OMG IDL compliant, the keylist definition is specific to OpenSplice and mandatory for all data types that are to be used as a topic. The OpenSplice preprocessor will not generate appropriate DataReaders and DataWriters for data types that do not have a corresponding keylist definition. A keylist definition should always be located in the same module as the data type it applies to. Apart from that requirement, the exact location of the keylist statement is irrelevant (it may be located before or after the actual definition of the data type).

Data Types without a keylist definition can still be used as embedded structures for data types that do have a keylist definition. Data Types that are to be used as topics but that do not require any keyfields (so called singleton instances) still require a keylist definition, but with an empty keylist. In case of the example above, if we did not require any keys, line *10* could be replaced by the following statement:

```
#pragma keylist ChatMessage
```

In the example above, only a very limited subset of IDL is being used. Apart from the trivial primitives (e.g. structures consisting of (unsigned) short, (unsigned) long, (unsigned) long long, float, double, boolean, octet and char), OpenSplice is also capable of handling fixed length arrays, bounded and unbounded sequences, bounded and unbounded strings, union types and enumerations. Types can be nested, which means that a struct can contain a struct field or an array of structs, or a

sequence of strings or an array of sequences containing structs or... many more complex examples you can think of. Any definition following the OpenSplice IDL subset is allowed (refer to the *OpenSplice DDS IDL Preprocessor Guide*). It is important to know that the preprocessor used by the DCPS accepts struct definitions only, not interfaces or value types (occurrences of both types will be ignored by this preprocessor[1]).

You have to remember, however, that in the case of `sequences` and `strings`, you as a programmer are responsible for claiming and releasing memory resources and initializing the data type. For example, the string field `content` of the `ChatMessage` can be used only after the programmer has allocated the necessary memory. For more information on using the generated C structs see the OMG's *C Language Mapping Specification.*

3.3 Language Specific Representation

Even though the data type is defined using IDL, your application (when written in C) will be using an equivalent C struct. This is achieved by invoking the OpenSplice DDS IDL preprocessor, an application that translates your IDL data type definition into a matching C definition. The exact translation is defined by the OMG IDL to C mapping. The ChatMessage definition will result in the following C code:

```
18   #include <dds_dcps.h>
19
20   #ifndef _Chat_ChatMessage_defined
21   #define _Chat_ChatMessage_defined
22   #ifdef __cplusplus
23   struct Chat_ChatMessage;
24   #else /* __cplusplus */
25   typedef struct Chat_ChatMessage Chat_ChatMessage;
26   #endif /* __cplusplus */
27  #endif /* _Chat_ChatMessage_defined */
28  Chat_ChatMessage *Chat_ChatMessage__alloc (void);
29
30  struct Chat_ChatMessage {
31     DDS_long userID;
32     DDS_long index;
33     DDS_string content;
34  };
```

As can be seen, the preprocessor alters the IDL typename by adding the prefix `Chat_` (generated from the IDL module name), to allow for the scoping required by the IDL module. It also provides a typedef named `Chat_ChatMessage`, which simplifies the declaration of a chatmessage variable because of its implicit struct declaration, as can be seen from the following example application:

1. In contrast, the DLRL preprocessor is able to handle value types. If your application needs to distribute information using valuetypes, consider using the DLRL for that purpose.

```
35    // explicit struct declaration.
36    struct Chat_ChatMessage_s message1;
37
38    // implicit struct declaration.
39    Chat_ChatMessage message2;
```

For C++, this way of declaring variables is already supported (so the typedef is not applied when a C++ compiler is being used), but for convenience we added it to the C API as well. The preprocessor also generates an allocation function, as mandated by the IDL to C language mapping, which can be used to allocate samples of a data type on heap. For our current example this allocation function is named `Chat_ChatMessage__alloc()` (see line *28*). Additional information is provided in the *OpenSplice DDS C Reference Guide*.

The type of each of the fields in the struct is based on the IDL to C mapping, with the difference that the `CORBA_` prefix of each primitive type is replaced by a `DDS_`

prefix. (The semantics for each of the types have not been changed with respect to the language mapping). This deviation represents the fact that we are dealing with a standalone C API, that has no dependencies on CORBA whatsoever. API's that cohabitate with CORBA use the pre-processor that comes with the ORB to do the IDL translation. In that case there will be plenty of CORBA dependencies in the generated code.

3.4 Invoking the IDL Pre-processor

If you want to reproduce the example, create a file named Chat.idl. Insert the IDL definition given in the previous example into this file. Run the IDL pre-processor from the command line using:

```
%  idlpp -S -l c Chat.idl
```

If it successfully completes, examine the resulting file called *`ChatDcps.h`*, which contains the C structs. Do not include this file directly into your application though, but use the `Chat.h` file instead. That file is a collection of all relevant information for your application. For now, ignore all other files that are also generated by the preprocessor, we will get back on some of those in a later section.

The ***`-S`*** option specifies that the IDL pre-processor should run in *StandAlone* mode, meaning that it does not have any dependency on CORBA and so can be used without any ORB being installed.

The ***`-l`*** option indicates the target language, which in this case represents C code. Other supported languages are Java (*`-l java`*) and C++ (*`-l cpp`*). See the *IDL Pre-processor Guide* for a summary of all other possible options.

CHAPTER

4 Managing Domains and Topics

In this section you will write your first OpenSplice DDS application. Before you are ready to start writing the first lines of code, we need to explain a little about some basic DDS building blocks and the way data is handled in OpenSplice DDS. The first example of an OpenSplice application is small and is just a declaration of the Domain to use, the topics to use inside it and the QoS settings that need to be applied to both.

The first section will introduce the generic API building blocks and explain their purpose. The second section will introduce you to the concept of QoS policies and will show the policies which are most relevant to our Chatter application. The third section will show you how to connect your application to a specific DDS Domain. The fourth section will demonstrate the steps that are necessary to introduce the required topics into that Domain.

4.1 Entities, Policies, Listeners and Conditions

The DDS can be seen as a large toolbox full of different building blocks. To understand the granularity of these DDS building blocks and the way in which they interact, we will first explain some higher level DDS concepts in more detail:

- ***Entity*** - An Entity is a basic DCPS building block. It represents either a producer of information (Publisher or DataWriter), a consumer of information (Subscriber or DataReader), a connection to information (DomainParticipant) or the information that is being communicated (Topic). The behaviour of each Entity can be influenced by means of QoS Policies that must be associated to it at creation time. To keep track of the communication status of an Entity, a StatusCondition object can be obtained from it, or a Listener object can be attached to it. An Entity can only be created or deleted using its corresponding factory. Some Entities may act as a factory for other Entities.
- ***QoS Policy*** - QoS Policies provide a generic mechanism for the application to control the behaviour of an Entity: each policy controls one aspect of the Entity and is represented by a structured type containing attributes for all relevant parameters. Entities have a varying set of supported policies: some of them are applicable to only one Entity, some others to more. To make sure neither more nor less than the supported policies are attached to each specific Entity, each Entity provides a specialized QoS structure that aggregates all applicable policies.

- ***StatusCondition*** - A StatusCondition object provides a generic mechanism for the application to be informed about relevant status changes in Entities, such as the availability of data corresponding to a subscription, conflicting QosPolicy settings between related Entities, contracts that are being violated, etc. Each of these individual statuses can be either TRUE or FALSE, and may change independently from all the others. The application can make a selection of the statuses it is interested in by setting a bit mask in the StatusCondition object, and when one or more of the selected statuses is TRUE, the overall status flag in the StatusCondition object itself becomes TRUE as well. This flag remains TRUE, until each and every of the selected statuses has been reset to FALSE again. Resetting these individual statuses can be done by invoking their corresponding status accessor method in the related Entity object. To find out which individual statuses are responsible for raising the StatusCondition flag, the Entity object offers a helpful operation that returns a mask that specifies the statuses that are currently set to TRUE.
- ***WaitSet*** - An application can use a WaitSet to block the current thread until one or more of the (Status) Conditions attached to that WaitSet will have a trigger value of TRUE, or until a specified timeout expires.
- ***Listener*** - A Listener provides a generic mechanism for the middleware to notify the application of changes in StatusConditions. Each Entity supports its own specialized kind of Listener interfaces, which offer specialized callback methods for every individual status change. The application can make a selection of the status changes it is interested in by setting a bit mask that can be supplied at creation time, or in the `set_listener` operation.

Although `DDS_Listeners` and `DDS_WaitSets` both allow the middleware to notify the application of the occurrence of certain events (so that it does not need to poll for this) there are two differences in their intended usage:

1. Listeners are event based and trigger only when a selected status flag changes from `FALSE` to `TRUE`. WaitSets are state based and will trigger as long as a selected status flag remains `TRUE`.
2. Listeners offer callback methods that are invoked by a middleware thread. This means that using Listeners always result in multi-threaded applications. WaitSets can be used to block the current application thread temporarily, and do not necessarily require your application to be multi-threaded.

If an application chooses to use both Listeners and WaitSets to be notified of status conditions in the same `DDS_Entities`, then OpenSplice will first trigger the `DDS_Listeners`, and after that (if the `DDS_StatusConditions` have not yet been reset by the listener operations) it will trigger the `DDS_WaitSets`.

4.2 QoS Policies

The way OpenSplice DDS communicates and stores samples, either in main memory or on disk, is defined by the key fields of their corresponding data type and the Quality of Service (QoS) Policies of their corresponding topic. Every topic must be created before it can be distributed by specifying its data type and associating a QoS Policy.

The QoS Policies that need to be associated with a specific topic describe several aspects of data management for that specific topic. In this tutorial we will not discuss each individual policy, but simply focus on the two most important ones, that define to a large extent the delivery characteristics of each participating Entity.

The Topic related QoS Policies that will be discussed in this tutorial are:

- ***DURABILITY*** - OpenSplice DDS supports four types of durability. DURABILITY defines the lifespan of the data, categorized into VOLATILE, TRANSIENT_LOCAL, TRANSIENT and PERSISTENT data. OpenSplice realizes no backup storage for volatile data. When volatile data is delivered, no guarantee is given that this data can be obtained again. Transient data is recorded by OpenSplice for late joining readers, but only during the up time of the OpenSplice infrastructure. As long as the OpenSplice infrastructure is up-and-running, a copy of all transient data is preserved. Persistent data outlives the lifetime of the OpenSplice infrastructure because it is saved on a number of redundant disks (depending on your configuration). Therefore a copy of persistent data is always available, even when the OpenSplice infrastructure is restarted. Typically, your system configuration data will be persistent. It is not wise to mark frequently updated information as PERSISTENT, since the benefits will probably not outweigh the overhead.
- ***RELIABILITY*** - Two types of RELIABILITY can be used in OpenSplice, which are BEST_EFFORT and RELIABLE delivery. Data that is annotated for a reliable delivery is guaranteed to arrive ultimately because of automatic re-transmission of lost samples. Data that is marked for a best effort delivery gives no more guarantees than the network does: it remains unnoticed when the data gets lost on its way. Choosing not to re-transmit lost samples may be useful when data loses its accuracy quickly; second tries may unnecessarily use the infrastructure when more recent updates have already been sent.

All QoS policies have pre-defined (factory) settings. For the policies presented above, the default settings are depicted in *Table 2*.Refer to the *C Reference Guide* for all other policies and default settings.

Table 2 Default QosPolicy Settings

QoS Policy	Attribute	Value
DURABILITY	kind	DDS_VOLATILE_DURABILITY_QOS
RELIABILITY	kind	DDS_BEST_EFFORT_RELIABILITY_QOS
	max_ blocking_time	100 ms.

4.3 Connecting to a Domain

With the following steps you will be guided to write a small OpenSplice application. The goal of this application is to publish messages, but you start with opening a connection to an OpenSplice Domain and will later add the creation of the required topics.

```
1    /* CreateTopics.c */
2
3    #include "dds_dcps.h"
4    #include "Chat.h"
5
6    int
7    main (
8      int argc,
9      char *argv[])
10   {
11
12     DDS_DomainParticipantFactory    dpf;
13     DDS_DomainParticipant           dp;
14     DDS_DomainId_t                  domain = NULL;
15     DDS_ReturnCode_t                status;
16
17    /* Create a DomainParticipantFactory and a DomainParticipant */
18    /* (using Default QoS settings).                              */
19
20    dpf = DDS_DomainParticipantFactory_get_instance();
21    if (!dpf) {
22      printf("Creating ParticipantFactory failed!!\n");
23      exit(-1);
24    }
25    dp = DDS_DomainParticipantFactory_create_participant (
26      dpf,
27      domain,
28      DDS_PARTICIPANT_QOS_DEFAULT,
29      NULL,
30      DDS_ANY_STATUS);
31    if (!dp) {
32      printf("Creating Participant failed!!\n);
33      exit(-1);
34    }
35
36    /* Deleting the DomainParticipant */
37    status = DDS_DomainParticipantFactory_delete_participant(
38                  dpf, dp);
```

```
39    if (status != DDS_RETCODE_OK) {
40     printf("Deleting participant failed. Status = %d\n", status);
41     exit(-1);
42    };
43
44    /* Everything is fine, return normally. */
45    return 0;
46  };
```

This application is complete, and can be compiled and run. To do so, you need to add the location of the OpenSplice header files to your compiler's include path and link the result to the OpenSplice shared libraries. The location of the header files can be found (relative to the OpenSplice DDS installation directory) in the *include/dcps/C/SAC* subdirectory. The installation directory is specified in the `OSPL_HOME` environment variable, which should have been initialized when you executed the release.com script. The shared library files can be found in the subdirectory lib, and in this case you will need to link your application to the dcpssac library[1].

When the application has been successfully compiled and linked, you will need to start the OpenSplice infrastructure before executing your application. This is necessary because your application will try to setup a connection to a DDS Domain, which does not exist if the OpenSplice infrastructure is not up and running. The infrastructure can be started by issuing the following command:

```
%  ospl start
```

This command will launch all services specified in the configuration file that is identified by the OSPL_URI environment variable. The default configuration file that comes with OpenSplice is good enough for the examples in this tutorial.

To see whether the OpenSplice infrastructure is already up and running, issue the ospl list command, it will give you an overview of all instances of OpenSplice that are running on your node. To stop a specific instance of OpenSplice, issue the ospl stop command. It will detach all applications, stop the services and release all memory on your node.

Now start your newly created application. If it is correct, you will not get any error messages, but you will not notice anything else happening as well. Let's have a look at what happens at each code line that was presented above.

In line *3*, the file `dds_dcps.h` is included. This file contains all generic API calls of OpenSplice that are available. When dealing with reading or writing specific data types, typed reader/writer calls are also required to handle these data types. These

1. On a UNIX like platform this file is named libdcpssac.so, on the Windows platform it is named dcpssac.dll.

typed interfaces must be generated by the OpenSplice pre-processor, and the resulting output file must be included as well. This is already done in line *4*, although no typed interfaces are yet presented in this stage.[1]

In line *20* the `DDS_DomainParticipantFactory` instance is obtained. The `DDS_DomainParticipantFactory` is a singleton, meaning that there can only be one participant factory in each process. Obtaining the factory for the first time with the `DomainParticipantFactory_get_instance()` call implicitly instantiates it. Making this call at a later moment in time returns the already existing participant factory.

i Note that the `DomainParticipantFactory_get_instance()` function is not re-entrant, so it may only be called by one thread at a time. (See also Section 8.8, *Cleaning Up*, on page 96.)

In lines *21-24* it is checked whether the factory handle obtained above is actually a valid handle (i.e. does not represent a `NULL` pointer). ALWAYS CHECK THE VALIDITY OF HANDLES RETURNED BY FUNCTION CALLS! Not doing so may result in failing function calls later on in your application, which are not easy to trace back to their root cause.

In lines *25-30* `DDS_DomainParticipantFactory_create_participant()` is invoked to create a `DDS_DomainParticipant`, which represents our connection to a specific DDS Domain. The first parameter for this operation (as for any DDS operation) represents the entity that actually needs to execute the function call, which in this case is our participant factory. The second parameter is the domain ID (represented by a URI - a Universal Resource Indicator), which should point to the OpenSplice configuration file containing the Domain related properties for this node. See the *OpenSplice DDS Deployment Guide* for additional information.

Not assigning a value to this URI, like we do in our example, means that OpenSplice will look for the configuration file in the location specified by the environment variable called `OSPL_URI`. (For Linux/Unix based platforms, this variable is initialized by sourcing the `release.com` script that is created by the OpenSplice installer. On Windows platforms, this variable is already initialized in your environment by your Windows installer. The variable will point to a default configuration file that comes with OpenSplice.). For our tutorial example that is okay for now.

The third parameter specifies the QoS settings that will be used for the `DDS_DomainParticipant`. Since we are satisfied with the pre-defined (factory) settings for the participant QoS, we indicate that we want to copy these factory settings (as is) to our `DDS_DomainParticipant` by using a so called convenience

1. In fact, the IDL preprocessor creates more files than just this one, but the file presented here is the one that includes all the other files that are relevant for the application.

macro. The DDS provides for each DDS_Entity a corresponding convenience macro that represents the default QoS for that DDS_Entity[1]. The name of that macro always consists of the prefix DDS_ followed by the name of the DDS_Entity (in the case of a DDS_DomainParticipant this name is shortened to PARTICIPANT), followed by the postfix _QOS_DEFAULT. This macro can be used at any location where a QoS for the corresponding Entity needs to be supplied by the application.

The last two parameters specify a DDS_DomainParticipantListener object that can be attached to the DDS_DomainParticipant and a bit mask identifying the status events on which it should trigger. In this example we are not interested in handling any status changes on the DDS_DomainParticipant, so we choose not to attach a listener object here. We do that by providing a NULL pointer for this parameter[2]. The bit mask specifies which status events should be handled by the supplied DDS_DomainParticipantListener object: each status is represented by a special constant that represents its bit position in the bit mask. See *Table 3* for an overview of the names and meaning of all these status events and the DDS_Entities to which they are applicable.

For all classes that inherit from DDS_Entity all events not handled by their attached listener objects will be propagated to the listener objects attached to their factories. Since we are not interested in propagating our events anywhere (we just want to ignore them) we select a bit mask that handles all appropriate events by our NULL listener[3]. There is a special constant called DDS_ANY_STATUS that selects all appropriate status bits for the DDS_Entity to which it is applied.

Table 3 Status Events Overview

DDS_Entity	Status Name	Meaning
DDS_Topic	DDS_INCONSISTENT_TOPIC_STATUS	Another DDS_Topic exists with the same name but with different characteristics.
DDS_Subscriber	DDS_DATA_ON_READERS_STATUS	New information is available.

1. There are convenience macros for other purposes as well.
2. A NULL listener behaves like a listener that handles all events it receives as a no-op.
3. A DDS_DomainParticipant has no factory to which it can propagate its events, so technically speaking it doesn't matter what bit-mask you select in this case. For all other DDS_Entities however it is an important consideration to make.

Table 3 Status Events Overview (Continued)

DDS_Entity	Status Name	Meaning
DDS_DataReader	DDS_SAMPLE_REJECTED_STATUS	A (received) sample has been rejected.
	DDS_LIVELINESS_CHANGED_STATUS	The liveliness of one or more DDS_DataWriter objects that were writing instances read through the DDS_DataReader has changed. Some DDS_DataWriter have become “active” or “inactive”.
	DDS_REQUESTED_DEADLINE_ MISSED_STATUS	The deadline that the DDS_DataReader was expecting through its DDS_DeadlineQosPolicy was not respected for a specific instance.
	DDS_REQUESTED_ INCOMPATIBLE_QOS_STATUS	A QosPolicy setting was incompatible with what is offered.
	DDS_DATA_AVAILABLE_STATUS	New information is available.
	DDS_SAMPLE_LOST_STATUS	A sample has been lost (never received).
	DDS_SUBSCRIPTION_MATCHED_STATUS	The DDS_DataReader has found a DDS_DataWriter that matches the DDS_Topic and has compatible QoS.
DDS_DataWriter	DDS_LIVELINESS_LOST_STATUS	The liveliness that the DDS_DataWriter has committed through its DDS_LivelinessQosPolicy was not respected; thus DDS_DataReader objects will consider the DDS_DataWriter as no longer “active”.
	DDS_OFFERED_DEADLINE_ MISSED_STATUS	The deadline that the DDS_DataWriter has committed through its DDS_DeadlineQosPolicy was not respected for a specific instance.
	DDS_OFFERED_INCOMPATIBLE_ QOS_STATUS	A QosPolicy setting was incompatible with what was requested.
	DDS_PUBLICATION_MATCHED_STATUS	The DDS_DataWriter has found DDS_DataReader that matches the DDS_Topic and has compatible QoS.

Table 3 Status Events Overview (Continued)

DDS_Entity	Status Name	Meaning
All DDS_Entity objects.	DDS_ANY_STATUS	All status events applicable to the DDS_Entity in question.

When the DDS_DomainParticipantFactory_create_participant operation completed successfully, it returns the handle to the created DDS_DomainParticipant. Again, check whether the resulting handle is valid before using it in other operations.

After the DDS_DomainParticipant has been created, the application is ready to use the OpenSplice infrastructure. The application could now create topics, publishers and subscribers, but we will do that in a later stage. For now, we will release the resources used by OpenSplice by deleting the participant again. We do this in the DDS_DomainParticipantFactory by invoking the DDS_DomainParticipantFactory_delete_participant() call. This operation deletes all resources used by this participant and returns a status code of type DDS_ReturnCode_t. Since we didn't do anything with our participant yet, the status code should indicate a successful result, represented by DDS_RETCODE_OK. However, never assume everything will go according to plan: always check your assumptions! In line *39* we check whether the result is what we expect. In a later stage, when our application has expanded a little bit, the result could indicate that we are not yet allowed to delete this participant. The possible return statuses of type DDS_ReturnCode_t are depicted in *Table 4*, together with their value and their meaning.

This concludes our first example. When you monitor all OpenSplice activity with the OpenSplice DDS Tuner, nothing seems to have happened. This is because the participant was created and deleted so fast, that OpenSplice DDS Tuner did not have the time to depict it. If you run the application in a debugger, and stop the execution before the DomainParticipantFactory_delete_participant() operation, then you will see that the OpenSplice DDS Tuner actually detects the DomainParticipant and shows it in its participant list. You can even check its QoS settings to see if they match the defaults that you specified. In a later example we will show you how you can provide your own QoS settings.

Table 4 Return Code Definitions

Return Code	Value	Meaning
DDS_RETCODE_OK	0	Successful return.
DDS_RETCODE_ERROR	1	Generic, unspecified error.
DDS_RETCODE_UNSUPPORTED	2	Unsupported operation. Can only be returned by operations that are optional.
DDS_RETCODE_BAD_PARAMETER	3	Illegal parameter value.
DDS_RETCODE_PRECONDITION_NOT_MET	4	A precondition for the operation was not met.
DDS_RETCODE_OUT_OF_RESOURCES	5	Service ran out of the resources needed to complete the operation.
DDS_RETCODE_NOT_ENABLED	6	Operation invoked on an Entity that is not yet enabled.
DDS_RETCODE_IMMUTABLE_POLICY	7	Application attempted to modify an immutable QosPolicy.
DDS_RETCODE_INCONSISTENT_POLICY	8	Application specified a set of policies that are not consistent with each other.
DDS_RETCODE_ALREADY_DELETED	9	The object target of this operation has already been deleted.
DDS_RETCODE_TIMEOUT	10	The operation timed out.
DDS_RETCODE_NO_DATA	11	Indicates a transient situation where the operation did not return any data but there is no inherent error.
DDS_RETCODE_ILLEGAL_OPERATION	12	An operation was invoked on an inappropriate object or at an inappropriate time (as determined by policies set by the specification or the Service implementation). There is no precondition that could be changed to make the operation succeed.

4.4 Registering Data Types and Creating Topics

We can now start using the `DDS_DomainParticipant` created in the previous example to actually create a `DDS_Topic`. Reiterating from the previous sections, a topic was an aggregation between a data type (including its key list) and a QosPolicy setting. So before being able to create a topic, first the corresponding data type will need to be registered in the middleware. To register a data type, we require

a some source code that announces the type specific meta data to OpenSplice. This code is embedded in a so called `DDS_TypeSupport` class, which is generated by the OpenSplice DDS Preprocessor.

The OpenSplice preprocessor generates a number of files out of each IDL input file. We already introduced two of these files:

- The file `Chat.h` is the overall include file. It includes all other files relevant for the application. Its name is based on the name of the corresponding IDL file, where the `.idl` extension is replaced by the `.h` extension.
- The file `ChatDcps.h` contains the C representations of the data structures defined in your IDL file. Its name is based on the base name of the corresponding IDL file, but it is appended by the postfix `Dcps.h`.

We will now explain a third file generated by the pre-processor, called `ChatSacDcps.h`. This file name is also based on the basename of the IDL file, but it is appended by the postfix `SacDcps.h` (Sac stands for Standalone C API, which is the OpenSplice DDS API that you are now using). It contains the specialized API interface definitions for the `DDS_TypeSupport`, `DDS_DataReader` and `DDS_DataWriter` classes parameterized for all data types mentioned in the IDL file[1]. It is a very big file, so we will not show it here entirely. Instead, we will focus on the parts that define the `DDS_TypeSupport` interface for our `ChatMessage` data type.

```
47   #include "ChatDcps.h"
48
49   #define Chat_ChatMessageTypeSupport DDS_TypeSupport
50
51   Chat_ChatMessageTypeSupport
52   Chat_ChatMessageTypeSupport__alloc (
53     void
54   );
55
56  DDS_ReturnCode_t
57  Chat_ChatMessageTypeSupport_register_type (
58    Chat_ChatMessageTypeSupport _this,
59    DDS_DomainParticipant domain,
60    DDS_string name
61  );
```

In line *47* we see that this file includes the C representations of the data types, which is necessary because the corresponding DataReaders and DataWriters will be accessing this data. Line *49* introduces the definition of our specialized `Chat_ChatMessageTypeSupport` class. Its name is based on the name of our

1. The corresponding ChatSacDcps.c file contains the implementation code for these interfaces.

data type (prepended by the `module` name in which it is located), and it is followed by the `TypeSupport` postfix. As can be seen from this declaration, the specialized TypeSupport handle is just an alias for the handle of its parent class.

Lines *51-54* present an allocation function that is needed to actually instantiate a TypeSupport object on heap. Its name is based on the specialized TypeSupport class, followed by the `__alloc()` postfix. Every DDS object allocated by an `__alloc()` operation must be released by using the `DDS_free()` operation, which is included from the `dds_dcps.h` file. Never try to de-allocate a DDS object any other way, since it will almost definitely corrupt your memory and crash your application.

Lines *56-61* finally present the operation required to register the data type in a `DDS_DomainParticipant`. This operation can only be performed on an allocated TypeSupport: forgetting to allocate the TypeSupport will probably result in a `DDS_RETCODE_BAD_PARAMETER`. A TypeSupport object may be registered in different `DDS_DomainParticipants`, but has no more purpose after the registering is completed, so it may be released afterwards. There is no way to un-register a data type, so after the `DDS_TypeSupport` has been released its registered data types can still be used in the `DDS_DomainParticipant`.

The `Chat_ChatMessageTypeSupport_register_type` method requires three parameters:

- the pointer to the allocated `Chat_ChatMessageTypeSupport` object
- the handle to the `DDS_DomainParticipant` in which it is to be registered
- the name by which this data type can be identified within the specified `DDS_DomainParticipant`

This name parameter is a little bit tricky, since it identifies the data type only in the scope of the specified `DDS_DomainParticipant`. Other participants could choose to register the same data type using a different name. This makes setting up communications between different DomainParticipants a hazardous task: what if two DomainParticipants have registered the same data type using different names?

To avoid such configuration problems, we advise you to always register a data type using its IDL type name. The `DDS_TypeSupport` offers helpful features for this:

- If you pass a `NULL` value to the name, the `DDS_TypeSupport` will register the data type using its IDL type name, including its scope, in other words. the names of the modules that the IDL data type is embedded in, separated by the IDL scoping operator, `::`. In this example the resulting name will be: `Chat::ChatMessage`.
- Alternatively, you can obtain the fully qualified IDL type name directly from a `DDS_TypeSupport` itself using the `Chat_ChatMessageTypeSupport_get_type_name()` operation in this example. The resulting name can then be used for both the registration of the type and the creation of the topic.

Using these tricks ensures you will always be using the same type name for a given data type in every DDS_DomainParticipant. We strongly advise you to always register the data types this way: only use different names when you have very compelling reasons to do so.

The data types registered this way can be used to create topics: the basic DDS communication entities. Creating a DDS_Topic is very similar to creating a DDS_DomainParticipant (remember that there are lots of similarities since both interfaces are specialiations of the DDS_Entity interface):

- A DDS_Entity can only be created and deleted by using its factory. The DDS_DomainParticipant acts as a factory for DDS_Topics.
- At creation time, a DDS_Entity needs to be associated with a set of QoS Policies.
- At creation time, a DDS_Listener can be attached to the entity, accompanied by a bit mask that indicates which status events need to be handled by the provided listener.

Below, we have expanded the example presented in Section 4.3, *Connecting to a Domain*, with the code that actually creates the ChatMessage topic:

```
62  /* CreateTopics.c */
63
64  #include "dds_dcps.h"
65  #include "Chat.h"
66
67  int
68  main (
69    int argc,
70    char *argv[])
71 {
72
73   DDS_DomainParticipantFactory    dpf;
74   DDS_DomainParticipant           dp;
75   DDS_DomainId_t                  domain = NULL;
76   DDS_ReturnCode_t                status;
77   Chat_ChatMessageTypeSupport     chatMessageTS;
78   DDS_Topic                       chatMessageTopic;
79   char                            *chatMessageTypeName;
80
81   /* Create a DomainParticipantFactory and a DomainParticipant */
82   /* (using Default QoS settings).                             */
83
84   dpf = DDS_DomainParticipantFactory_get_instance();
85   if (!dpf) {
86     printf("Creating ParticipantFactory failed!!\n");
87     exit(-1);
88   };
89   dp = DDS_DomainParticipantFactory_create_participant(
90     dpf,
91     domain,
92     DDS_PARTICIPANT_QOS_DEFAULT,
93     NULL,
94     DDS_ANY_STATUS);
```

```
95    if (!dp) {
96      printf("Creating Participant failed!!\n);
97      exit(-1);
98    };
99
100   /* Register the required data type for ChatMessage. */
101   chatMessageTS = Chat_ChatMessageTypeSupport__alloc();
102   if (!chatMessageTS) {
103     printf ("Allocating TypeSupport failed!!\n");
104     exit(-1);
105   };
106   chatMessageTypeName =
107     Chat_ChatMessageTypeSupport_get_type_name(chatMessageTS);
108   status = Chat_ChatMessageTypeSupport_register_type(
109     chatMessageTS, dp, chatMessageTypeName);
110   if (status != DDS_RETCODE_OK) {
111     printf (
112         "Registering data type failed. Status = %d\n", status);
113     exit(-1);
114   };
115
116   /*Create the ChatMessage topic */
117   chatMessageTopic = DDS_DomainParticipant_create_topic(
118     dp,
119     "Chat_ChatMessage",
120     chatMessageTypeName,
121     DDS_TOPIC_QOS_DEFAULT,
122     NULL,
123     DDS_ANY_STATUS);
124   if (!chatMessageTopic) {
125     printf("Creating ChatMessage topic failed!!\n");
126     exit(-1);
127   };
128
129   /* Deleting the Topic. */
130   status = DDS_DomainParticipant_delete_topic(
131     dp, chatMessageTopic);
132   if (status != DDS_RETCODE_OK) {
133     printf("Deleting topic failed. Status = %d\n", status);
134     exit(-1);
135   };
136
137   /* Deleting the DomainParticipant */
138   status = DDS_DomainParticipantFactory_delete_participant(
139     dpf, dp);
140   if (status != DDS_RETCODE_OK) {
141     printf("Deleting participant failed. Status = %d\n", status);
142     exit(-1);
143   };
144
145   /* Everything is fine, return normally. */
146   return 0;
147 };
```

As can be seen from this code example in lines *101-114*, a `Chat_ChatMessageTypeSupport` is allocated and its data type is registered in our `DDS_DomainParticipant` using its default name. Again, the result of every operation is checked against our assumptions.

In lines *117-127*, we create our first topic using the `DDS_DomainParticipant_create_topic()` operation. As always, the first parameter is the handle to the object that actually needs to perform the operation (our `DDS_DomainParticipant`). The second parameter provides the name that will be used to identify the topic. This is also the name that we will see when we display our topic list in the OpenSplice DDS Tuner. The third parameter is the name of the data type that we wish to associate with our topic. In our case, this is the default name provided by our `Chat_ChatMessageTypeSupport` class. The fourth, fifth and sixth parameters are the set of QoS Policies we wish to associate with the topic, the `DDS_TopicListener` we wish to attach to it and the bit mask which applies to that listener respectively. In this case we again used a convenience macro to select the default set of QoS Policies for this topic, and we also specified that we do not want to attach a Listener.

In this example, we don't use our topic for any purpose yet: we delete it just before we delete our `DDS_DomainParticipant`. This is necessary, since in the DDS it is not possible to delete any type of factory that still contains elements that are created by it. In our case, the `DDS_DomainParticipant` acted as a factory for our `DDS_Topic`, and can therefore not be deleted while our topic object still exists. Trying to delete the participant in this stage will definitely result in a `DDS_RETCODE_PRECONDITION_NOT_MET` being returned.

That is why we need to delete the topic first. This is done in line *130*, by means of the `DDS_DomainParticipant_delete_topic()` operation, whose parameter signature is very obvious and needs no further explanation. After the topic has been deleted, the `DDS_DomainParticipant` can be deleted without any problems as well. That ends our little application for now.

4.5 Topics as Global Concepts

When we look in the OpenSplice DDS Tuner at the results of the application presented in the previous section, we will see that although our DomainParticipant has disappeared, our topic is still available in the list of topics. This is not a bug! To understand what is happening here, we need to elaborate a little bit more on the global concept of a topic. A topic represents the smallest undividable part of an information model that can be communicated within a domain. In order for the communication to be successful, all parties within the domain must agree upon how the information is distributed and what it represents. That means that the topic definition is not just something local: all participants in our domain must agree upon it.

That means that if I create a topic in my DomainParticipant, this topic will automatically be forwarded to all other participants in my domain. They will then compare it to the topic definitions they already know. If my topic definition matches with already existing definitions or does not yet exist, my topic definition is

accepted and my call returns successfully. If my definition conflicts with an already existing topic definition, my creation will fail and my call will return a `NULL` pointer.

So the `DDS_Topic` I create is not just a local object it represents a global concept of a part of an information model, agreed upon by all parties within my domain. The Topic object I create is just a 'proxy' that represents this global concept. Deleting my local `DDS_Topic` object will not destroy it globally: it will merely destroy my local proxy that represents it. This seems natural: one party joining a system that already agreed upon its topic model, cannot by itself decide to destroy this global topic model when it decides to leave the system. It can only decide for itself that it is no longer interested in the existence of certain topics, without interfering with the parts of the system that still do want to know about them.

This is why a topic as a global concept cannot be deleted: you never know which parts of the system may still have a need for it. When you really want to completely remove a topic definition from a running system, you will need to bring down all applications joining in your domain, stop their daemons and restart everything. This is why you should be careful when introducing new topics into a running system: you cannot easily undo any mistakes you make. Creating new topics is therefore not something that everybody should be allowed to do: a system architect should be made responsible for defining an overall information model that all participants need to agree upon[1].

4.6 Tailoring QosPolicy Settings

In the previous examples we defined a complete information model for our Chatroom application in IDL, but we only created topics using default QoS settings. In this section we will elaborate on the requirements for our `Chatter` application, and decide which QosPolicy settings are best suitable for our application.

Since we do not want to lose any chat message or username, both topics will have to be transmitted reliably. A late joining chatter application is probably not interested in receiving the chat messages that were transmitted before he decided to join in, but it will definitely want to be able to figure out which `userID` represents which username once it starts to receive chat messages. That means that the `ChatMessage` topic can be transmitted with volatile durability, but the `NameService` topic will require either transient or persistent storage. Since chatter application will always publish its username before writing its chat messages, the storage of these names will not need to be persistent, and a transient store will be sufficient.

1. An individual application is however allowed to create its own local view of existing topics by using a MultiTopic. This can only be used for reading information, not for writing it.

For a late joining application this means that once it subscribes itself to the `NameService` topic, it will receive from the transient store the usernames and `userID`'s of all other Chatters that have already connected to the same Domain before. In contrast, it will only receive those chat messages that have been transmitted after its own subscription to the `ChatMessage` topic.

To make our Chatroom application work this way, we need to deviate from the default QoS settings. These default QoS Policies have been chosen in such a way that they form an internally consistent set that is most suitable to 'first time users' and that gives a good 'out of the box' experience. When dedicated requirements call for alternative QoS settings on your Entities, you can tailor these settings in the following ways:

1. You can create Entities using a QoS in which each policy is set explicitly.
2. You can obtain the default QoS, modify some of its policies to match your own preference, and use the result to create your Entities.
3. You can permanently make changes to the default QoS of each factory.

All these approaches have their benefits in certain conditions. You can explicitly set each policy when you need very dedicated settings that do totally not comply with the factory defaults. However, if you reuse the same settings in most of your Entities, it makes sense to use the default settings from your factories, even when you need to modify these factory defaults first. When you are satisfied with the default policies, but need little deviations from them every now and then, it makes sense to obtain the default QoS, modify some of the policies to fit a specific Entity, and create that Entity with it.

The following code again expands our example application, but this time we will create both topics using different QoS settings. The explicit checks on the validity of return statuses and handles have all been replaced with specialized functions, which are included from the `CheckStatus.h` file, and implemented in the `CheckStatus.c` file. The code listings for both files can be found under *CheckStatus.h* and *CheckStatus.c* in Appendix A, *C Language Examples' Code*:

```
148  /* CreateTopics.c */
149
150  #include "dds_dcps.h"
151  #include "Chat.h"
152  #include "CheckStatus.h"
153
154  int
155  main (
156    int argc,
157   char *argv[])
158 {
159   DDS_DomainParticipantFactory    dpf;
160   DDS_DomainParticipant           dp;
161   DDS_DomainId_t                  domain = NULL;
162   DDS_ReturnCode_t                status;
```

```
163   Chat_ChatMessageTypeSupport     chatMessageTS;
164   Chat_NameServiceTypeSupport     nameServiceTS;
165   char                            *chatMessageTypeName;
166   char                            *nameServiceTypeName;
167   DDS_TopicQos                    *reliable_topic_qos;
168   DDS_TopicQos                    *setting_topic_qos;
169   DDS_Topic                       chatMessageTopic;
170   DDS_Topic                       nameServiceTopic;
171
172   /* Create DomainParticipantFactory and a DomainParticipant */
173   /* (using Default QoS settings).                            */
174
175   dpf = DDS_DomainParticipantFactory_get_instance();
176   checkHandle(dpf, "DDS_DomainParticipantFactory_get_instance");
177   dp = DDS_DomainParticipantFactory_create_participant (
178     dpf,
179     domain,
180     DDS_PARTICIPANT_QOS_DEFAULT,
181     NULL,
182     DDS_ANY_STATUS);
183   checkHandle(
184     dp, "DDS_DomainParticipantFactory_create_participant");
185
186   /* Register the required data type for ChatMessage. */
187   chatMessageTS = Chat_ChatMessageTypeSupport__alloc();
188   checkHandle(
189     chatMessageTS, "Chat_ChatMessageTypeSupport__alloc");
190   chatMessageTypeName =
191     Chat_ChatMessageTypeSupport_get_type_name(chatMessageTS);
192   status = Chat_ChatMessageTypeSupport_register_type(
193     chatMessageTS, dp, chatMessageTypeName);
194   checkStatus(
195     status, "Chat_ChatMessageTypeSupport_register_type");
196
197   /* Register the required data type for NameService. */
198   nameServiceTS = Chat_NameServiceTypeSupport__alloc();
199   checkHandle(
200     nameServiceTS, "Chat_NameServiceTypeSupport__alloc");
201   nameServiceTypeName =
202       Chat_NameServiceTypeSupport_get_type_name(nameServiceTS);
203   Chat_NameServiceTypeSupport_register_type(
204     nameServiceTS, dp, nameServiceTypeName);
205   checkStatus(
206     status, "Chat_NameServiceTypeSupport_register_type");
207
208   /* Change the default TopicQos to Reliable reliability. */
209   reliable_topic_qos = DDS_TopicQos__alloc();
210   checkHandle(reliable_topic_qos, "DDS_TopicQos__alloc");
211   status = DDS_DomainParticipant_get_default_topic_qos(
212     dp, reliable_topic_qos);
213   checkStatus(
214     status, "DDS_DomainParticipant_get_default_topic_qos");
215   reliable_topic_qos->reliability.kind =
216                                   DDS_RELIABLE_RELIABILITY_QOS;
217
218   /* Make the tailored QoS the new default. */
219   status = DDS_DomainParticipant_set_default_topic_qos(
220     dp, reliable_topic_qos);
221   checkStatus(
222     status, "DDS_DomainParticipant_set_default_topic_qos");
223
```

```
224   /*Create the ChatMessage topic */
225   chatMessageTopic = DDS_DomainParticipant_create_topic(
226     dp,
227     "Chat_ChatMessage",
228     chatMessageTypeName,
229     DDS_TOPIC_QOS_DEFAULT,
230     NULL,
231     DDS_ANY_STATUS);
232   checkHandle(
233     chatMessageTopic,
234     "DDS_DomainParticipant_create_topic (ChatMessage)");
235
236   /* Obtain a private copy of the default QoS to tailor it. */
237   setting_topic_qos = DDS_TopicQos__alloc();
238   checkHandle(setting_topic_qos, "DDS_TopicQos__alloc");
239   status = DDS_DomainParticipant_get_default_topic_qos(
240     dp, setting_topic_qos);
241   checkStatus(
242     status, "DDS_DomainParticipant_get_default_topic_qos");
243
244   /* Note: changing the copy doesn't change the original
itself!*/
245   setting_topic_qos->durability.kind =
246     DDS_TRANSIENT_DURABILITY_QOS;
247
248   /* Associate the tailored policy with the NameService topic */
249   nameServiceTopic = DDS_DomainParticipant_create_topic(
250    dp,
251    "Chat_NameService",
252    nameServiceTypeName,
253    setting_topic_qos,
254    NULL,
255    DDS_ANY_STATUS);
256  checkHandle(
257    nameServiceTopic,
258    "DDS_DomainParticipant_create_topic (NameService)");
259
260  /* Deleting the Topics to be able to delete my participant. */
261  status = DDS_DomainParticipant_delete_topic(
262    dp, nameServiceTopic);
263  checkStatus(
264    status,
265    "DDS_DomainParticipant_delete_topic (NameServiceTopic)");
266
267  status = DDS_DomainParticipant_delete_topic(
268    dp, chatMessageTopic);
269  checkStatus(
270    status,
271    "DDS_DomainParticipant_delete_topic (chatMessageTopic)");
272
273  /* De-allocate the QoS policies. */
274  DDS_free(reliable_topic_qos);
275  DDS_free(setting_topic_qos);
276  DDS_free(pub_qos);
277
278  /* De-allocate the type-names and TypeSupports. */
279  DDS_free(nameServiceTypeName);
280  DDS_free(chatMessageTypeName);
281  DDS_free(nameServiceTS);
282  DDS_free(chatMessageTS);
```

```
283
284   /* Deleting the DomainParticipant */
285   status = DDS_DomainParticipantFactory_delete_participant(
286     dpf, dp);
287   checkStatus(
288     status,
289     "DDS_DomainParticipantFactory_delete_participant");
290
291   /* Everything is fine, return normally. */
292   return 0;
293 };
```

This example starts like the previous ones, but in line *209* we allocate a holder for the `DDS_TopicQos` that we will be using to create our topics. Since the change we want to make to our TopicQos is only minor compared to the default TopicQos, we will not set each policy field explicitly, but instead in line *211* we request the `DDS_DomainParticipant` to fill our holder with the current values of the default Topic Qos. Now we only have to change explicitly those QoS fields in the holder that are not suitable for our application. For our first topic, only the RELIABILITY settings will need to be changed and this is done in line *215*. Since all other topics that we will create in this `DDS_DomainParticipant` also require reliable transportation, it makes sense to make this the new default setting for this participant. (Note: default QoS settings are a property of the factory: different factories can have different default settings!). The participant default is changed according to the settings specified in our holder in line *219*.

The creation of the `ChatMessage` topic now in lines *225-231* is not really different from its creation it in the previous example, but since we changed the default QoS, the resulting topic will be different as well. If you did not restart your OpenSplice daemons after running the previous example, the creation of the current topic will fail since its QoS settings conflict with the settings of the previous example. In the OpenSplice DDS Tuner you will now be able to see that the `ChatMessage` topic indeed has different QoS settings and will be transported reliably.

The `NameService` topic requires another QoS change, so we will use the same trick employed before. This time however, since it is the only topic that requires transient durability, we will not change the default, but just create a custom QoS holder that we adapt to our needs. Again we fill it with the default QoS settings in line *239*, but this time we change the durability field to TRANSIENT durability in line *245*. We can now use our customized QoS holder in the creation of the `NameService` topic in lines *249-255*.

Don't forget to de-allocate your QoS holders, type-names and TypeSupport objects when you no longer need them. In our case, this is performed in lines *274-282*. Remember: the `DDS_free` operation can and must be used on any handle that was obtained by an operation whose name end with `__alloc()`, and on any string that is allocated as a result of a getter-operation on an entity.

This ends our first application now. We have shown you how to define an information model that suits your needs, how to select an efficient QoS that fits this model and how to create topics according to these choices. In the coming sections we will show you how to use these topic definitions to publish information into the system, and how to access this information in other applications by making subscriptions to these topics.

CHAPTER

5 Publishing the Data

In this section, you will be guided to create the publishing part of the chatter application. You will use the topic definitions of the previous section to publish your username and userID into the chatter domain, send an arbitrary number of chat messages afterwards, and then indicate that you leave the chatroom by disposing your username and ID.

The first section will give a short explanation of the different DDS entities that play a role in the publishing part of an application. The next section will teach you how to create a Publisher with accompanying DataWriters. That is followed by a section that describes the principles behind RxO QosPolicy matching between Readers and Writers and a section that describes how to delete your Publishers and Subscribers. The last two sections will show you how to use a DataWriters to register instances, write data samples into the system, and how to unregister and dispose these instances afterwards.

5.1 Publishers, DataWriters and their QoS Policies

Publishers and DataWriters are the building blocks required to publish information into your system. Both classes are modelled as `DDS_Entities`, meaning both are controlled by a set of QoS Policies, both have their own `DDS_StatusCondition`, both classes can have their own `DDS_Listener` object attached to them, and both classes can only be created and deleted by means of their corresponding factories. This section will introduce the reasons for separating Publishers from DataWriters in the DDS specification and explain the different objectives of both entities.

- ***Publisher*** - A Publisher is responsible for the dissemination of publications, in other words, the Publisher decides what information is to be published at what time and in which partition. The Publisher's QoS policiescontrol whether samples will be transmitted individually or as coherent sets of information (in order to allow for some primitive form of Transactions), whether the ordering between them will be preserved, and in which Partitions the information will be made available. The DomainParticipant acts as a factory for Publishers.
- ***Partition*** - The Partition QoSPolicy defines in which partitions information will be made available. Partitions are identified by name, and allow you to logically partition your information space: only when a publisher and a subscriber are connected to the same partition, communication will be established[1]. The PartitionQoSPolicy consists of an unbounded sequence of strings: each element

represents the name of a partition to which you will be connected. Elements containing names that have not yet been used before result in the creation of new Partitions. Elements may also contain wildcards, which will then be matched against all existing Partitions.

- ***DataWriter*** - A DataWriter is a type specific interface for the Publisher, in other words, it allows an application to offer samples for a specific topic to the Publisher, which will then perform the actual transmission of these samples. A Publisher acts as a factory for its own set of typed DataWriters, and can publish information that spans more than one Topic. In such cases, it employs a separate DataWriter for each individual Topic. The QoS Policies of a DataWriter control how its samples will be transmitted by the Publisher (e.g. their reliability and durability settings).

As you might have noticed from the previous bullet, some of the QoS Policies that you need to specify on the DataWriter are already specified on the Topic as well. That means that you might have conflicting QoS settings for a Topic on one hand, and for the DataWriters that actually provide samples for that specific Topic on the other hand. You might wonder why the DDS specification introduces such QoS Policy overlaps.

The reason is quite simple: the Topic QoS Policies act as some sort of system preference for all DataWriters (and also all DataReaders) of that Topic in your system. Normally, the system architect will select the most appropriate QoS Policiy settings that should be applicable to most DataReader/DataWriter combinations in your system, and he will attach those QosPolicy settings to the Topic. If you, as an application programmer, do not know what policies to use on your DataWriters (or DataReaders), just use the policies specified on the Topic.

However, you as an application programmer may have a very good reason to deviate from this system preference because of some dedicated knowledge you have about the behaviour of your application. In such cases you can tailor the DataWriter QoS Policy settings to your own needs, since it is always the QosPolicy settings on each individual DataWriter that decide how the samples are being transmitted.

1. You can also partition your information space by using different Domains (physical partitioning), which is a very static approach since an application cannot easily change the Domain it is attached to. In contrast, logical partitioning allows you to change your region-of-interest on the fly: you can change the number and type of partitions you are attached to at any moment in time.

5.2 Creating Publishers and DataWriters

In this section we will expand the example presented in Section 4.6, *Tailoring QosPolicy Settings*, with some code that creates our `DDS_Publisher` together with its two `DDS_DataWriters`: one for the `NameService` Topic, and one for the `ChatMessage` Topic.

The following code fragment shows the code fragments that should be inserted (between lines *258* and *260*) in order to create the `DDS_Publisher` with its `DDS_DataWriters` (it does not show the code already provided under *Tailoring QosPolicy Settings*.

```
1   DDS_PublisherQos                  *pub_qos;
2     DDS_DataWriterQos               *dw_qos;
3     DDS_Publisher                   chatPublisher;
4     Chat_ChatMessageDataWriter      talker;
5     Chat_NameServiceDataWriter      nameServer;
6     char                            *partitionName = NULL;
7
8     /* Adapt the default PublisherQos to write into the
9        "ChatRoom" Partition. */
10    partitionName = "ChatRoom";
11  pub_qos = DDS_PublisherQos__alloc();
12  checkHandle(pub_qos, "DDS_PublisherQos__alloc");
13  status = DDS_DomainParticipant_get_default_publisher_qos (
14    participant, pub_qos);
15  checkStatus(
16    status, "DDS_DomainParticipant_get_default_publisher_qos");
17  pub_qos->partition.name._length = 1;
18  pub_qos->partition.name._maximum = 1;
19  pub_qos->partition.name._buffer = DDS_StringSeq_allocbuf (1);
20  checkHandle(
21    pub_qos->partition.name._buffer, "DDS_StringSeq_allocbuf");
22  pub_qos->partition.name._buffer[0] = DDS_string_alloc (
23    strlen(partitionName));
24  checkHandle(
25    pub_qos->partition.name._buffer[0], "DDS_string_alloc");
26  strcpy (pub_qos->partition.name._buffer[0], partitionName);
27
28  /* Create a Publisher for the chatter application. */
29  chatPublisher = DDS_DomainParticipant_create_publisher(
30    participant, pub_qos, NULL, DDS_ANY_STATUS);
31  checkHandle(
32    chatPublisher, "DDS_DomainParticipant_create_publisher");
33
34  /* Create a DataWriter for the ChatMessage Topic
35     (using the appropriate QoS). */
36  talker = DDS_Publisher_create_datawriter(
37    chatPublisher,
38    chatMessageTopic,
39    DDS_DATAWRITER_QOS_USE_TOPIC_QOS,
40    NULL,
41    DDS_ANY_STATUS);
42  checkHandle(
43    talker, "DDS_Publisher_create_datawriter (chatMessage)");
44
45  /* Create a DataWriter for the NameService Topic
```

```
46      (using the appropriate QoS). */
47    dw_qos = DDS_DataWriterQos__alloc();
48    checkHandle(dw_qos, "DDS_DataWriterQos__alloc");
49    status = DDS_Publisher_get_default_datawriter_qos(
50      chatPublisher, dw_qos);
51    checkStatus(
52      status, "DDS_Publisher_get_default_datawriter_qos");
53    status = DDS_Publisher_copy_from_topic_qos(
54      chatPublisher, dw_qos, setting_topic_qos);
55    checkStatus(status, "DDS_Publisher_copy_from_topic_qos");
56    dw_qos->writer_data_lifecycle.autodispose_unregistered_instances =
57      FALSE;
58    nameServer = DDS_Publisher_create_datawriter(
59      chatPublisher,
60      nameServiceTopic,
61      dw_qos,
62      NULL,
63      DDS_ANY_STATUS);
64    checkHandle(
65      nameServer, "DDS_Publisher_create_datawriter (NameService)");
```

As you can see, in lines *11-14* a holder for the `PublisherQos` is allocated on heap and the default QosPolicy settings are copied into it. In lines *17-26*, the PartitionQosPolicy value is changed from its default value into a user defined Partition called *`ChatRoom`*. It is interesting to elaborate a little bit more on this, since besides demonstrating the Partition mechanism it also shows how to use IDL `sequences` and `strings` in the C language mapping.

As stated before, the PartitionQosPolicy is a sequence of strings. The default policy value is a sequence of zero elements, which is interpreted as a connection to the default Partition[1]. To attach to our own user defined Partition, we first need to allocate elements for the Partition sequence. A sequence in C is mapped onto a structure that contains a number of attributes:

- A field named `_maximum`: indicates the number of allocated elements.
- A field named `_length`: indicates the number of assigned elements.
- A field named `_buffer`: indicates a pointer to the first element.

In order to connect to only one Partition, we will need to allocate and assign at least one element. That means that the `_maximum` and `_length` fields can be set to 1, and that the `_buffer` field should point to a memory location that is able to hold a pointer to a string. The easiest way to allocate sequence elements is to use the convenience function that is generated by the OpenSplice DDS preprocessor specifically for that purpose. It is named after the sequence type (in this case `DDS_StringSeq`), followed by the postfix `_allocbuf`. Its parameter specifies the number of elements that need to be allocated.

1. The name of this default Partition is an empty string (""), so a Partition-sequence of 0 elements is equal to a Partition sequence of 1 element with an empty string.

In line *22* we actually allocate the memory for the `ChatRoom` string itself, using another dedicated function provided by the DDS API: `DDS_string_alloc`, where the parameter specifies the number of bytes to allocate[1]. The functions used to obtain the string length and to copy string contents are included from the standard `string.h` library. The reason why we use our own allocation functions instead of the more common `malloc` and `free` will become clear when we will release the memory later on.

Now that the `PublisherQos` has been tailored to our own needs, we invoke the `DDS_DomainParticipant_create_publisher` function in line *29*, to instruct the `DDS_DomainParticipant` (1st parameter) to create a new `DDS_Publisher` using our tailored QoS (2nd parameter) and no `DDS_PublisherListener` for all status events (3rd and 4th parameter). Again, the result is checked for correctness in line *31*.

In line *36*, we invoke the `DDS_Publisher_create_datawriter` function to instruct the `DDS_Publisher` (1st parameter) to create a typed DataWriter for the `chatMessageTopic` (2nd parameter) with QosPolicy values that are copied directly from the corresponding `DDS_TopicQos` (3rd parameter) and no `DDS_DataWriterListener` for all status events (4th and 5th parameter). The third parameter we used is again an example of a convenience macro: it is a substitute for a number of explicit steps, which would normally be:

- Allocate a `DDS_DataWriterQos` holder (`DDS_DataWriterQos__alloc`)
- Fill it with the default `DDS_DataWriterQos` settings of the DomainParticipant (`DDS_DomainParticipant_get_default_datawriter_qos`)
- Overwrite the policy values that overlap with the corresponding `DDS_TopicQos` by the values of that `DDS_TopicQos` (`DDS_Publisher_copy_from_topic_qos`).

In lines *47-55* an example of setting the `DDS_DataWriterQoS` using these explicit steps is shown. In this case, we do not use the convenience macro because we want to make one small modification to the resulting QoS (see lines *56-57*): we want to change the `writer_data_lifecycle` QosPolicy so that the `nameServer` does not automatically dispose a username when the user leaves the chatroom, which is its default behaviour. The exact meaning of this QosPolicy setting will be explained in Section 5.6, *Unregistering and Disposing of Instances*.

1. The `DDS_string_alloc` function allocates one more byte to accommodate for the '`\0`' terminator as well.

5.3 Requested/Offered QosPolicies Semantics

If the QosPolicies that are applicable to the DataWriter are closely examined, it will be observed that some of these policies overlap with the policies applicable to the topic. That is the `DDS_Publisher_copy_from_topic_qos` function is used to match all overlapping QosPolicies between topic and DataWriter.

Why do some of these policies overlap and what happens if they do not match? Before explaining the underlying mechanisms, let's first take a look at *Table 5*, which gives an overview of all QosPolicies that are applicable to Topics, DataWriters and DataReaders:

Table 5 Applicable Topic, DataWriter and DataReader Policies

QoS Policy	Concerns	RxO
DURABILITY	Topic, DataWriter, DataReader	Yes
DEADLINE	Topic, DataWriter, DataReader	Yes
OWNERSHIP	Topic, DataWriter, DataReader	Yes
LIVELINESS	Topic, DataWriter, DataReader	Yes
RELIABILITY	Topic, DataWriter, DataReader	Yes
DESTINATION_ORDER	Topic, DataWriter, DataReader	Yes
HISTORY	Topic, DataWriter, DataReader	No
RESOURCE_LIMITS	Topic, DataWriter, DataReader	No

In some of these cases, the QosPolicy settings are local to an entity and do not affect the behaviour of other (related) entities. Examples of these are `HISTORY` and `RESOURCE_LIMITS`, that specify how much storage space an entity reserves for buffering samples. In those situations, the `DataWriterQos` specifies how much storage space is reserved in the DataWriter and the `DataReaderQos` specifies how much storage space is reserved by the DataReader. DataWriters and DataReaders can make different choices without affecting each other's behaviour.

In the other cases, QosPolicy settings are not local to an entity and the DataReader and DataWriter will need to agree on the QosPolicy settings in order to establish successful communication. If the QosPolicies are considered compatible, then the DataWriter and DataReader will establish a successful connection. If the QosPolicies are considered incompatible, then the DataWriter and DataReader will be disconnected and not be able to communicate.

So when are policy settings considered compatible? That is decided by means of a subscriber-Requested/publisher-Offered (RxO) pattern. In this pattern, the DataReader can specify a *requested* value for a particular QosPolicy, while the DataWriter can specify an *offered* value for that QosPolicy. The Service will then determine whether the value requested by the DataReader is not considered 'higher'

than what is offered by the DataWriter. For this purpose, each RxO enabled Qospolicy will specify an ordering between its possible values to be able to make a comparison and determine the higher value. As long as the requested value is considered smaller than or equal to the offered value, the policies are considered compatible. If the requested value is higher than the offered value, the policies are considered incompatible, and the concerned DataWriter will raise an `OFFERED_INCOMPATIBLE_QOS` status, while the concerned DataReader will raise its `REQUESTED_INCOMPATIBLE_QOS` status. The application can detect this status change by means of a `Listener` or a `StatusCondition` (see Section 7.3.2, *Attaching a Listener* and Section 8.3, *Using a StatusCondition*).

Take as an example the `ReliabilityQosPolicy`: `RELIABLE` communication is considered *better* than `BEST_EFFORT` communication and so it has a higher value. A DataWriter that offers `BEST_EFFORT` communication will not attempt to retransmit samples that are lost, and so cannot satisfy the reliability request of a DataReader. In that case the requested value is higher than the offered value so the DataWriter and DataReader will be considered incompatible and can not communicate. However, a DataReader that requests `BEST_EFFORT` communication can be connected to a DataWriters that offers `RELIABLE` data, since the quality of the data that it gets is 'better' than what it required. In that case the requested value is lower than the offered value and so the policies are considered compatible.

Likewise for the DurabilityQosPolicy, the ordering of the possible values is `PERSISTENT > TRANSIENT > TRANSIENT_LOCAL > VOLATILE`. All other QosPolicies are outside the scope of this tutorial, so for the ordering of their QosPolicy values please consult the Reference Manuals.

So now it is clear what happens when you set different QosPolicy values on DataReaders and DataWriters, but how exactly do they relate to the QosPolicy values set on the Topic? To answer that question, it is important to realize what the QosPolicy settings on each Entity actually represent:

- The QosPolicy settings on a DataWriter define the amount of quality used to transport each sample written by that DataWriter.
- The QosPolicy settings on a DataReader define the requirements for the minimal amount of quality that each of the received samples should have. Samples that are transmitted with a lower quality will not be received.
- The QosPolicy settings on the Topic focus on global information-availability aspects rather than transmission-aspects of individual applications and represent the intended system behaviour.

Typically the information model is defined by a system architect, whose job is not only to think about the information content, but also about the Quality of Service that is normally required to transmit this information with. So he is responsible for designing an overall Topic model, which is an aggregation of datatypes and `TopicQos` settings.

The applications are typically designed by application developers, who will define all required publications and subscriptions, including the `DataWriterQos` and `DataReaderQos` settings. In normal circumstances, they will just copy the QosPolicy settings from the Topics, since those contain the settings as they are intended by the System Architect. Only in very special circumstances should an Application Developer deviate from `TopicQos` settings, for example when he knows that the samples he will read or write require different treatment than the rest of the samples of the same topic. Be careful with deviating from the TopicQos settings though, there is a good chance you will get disconnected from most of the other DataWriters or DataReaders who do follow the `TopicQos`.

Summarizing: the `TopicQos` specifies the QosPolicy settings the system architect intends the samples to be transmitted with, and so makes a good default setting for your DataWriters and DataReaders. However, deviating from the `TopicQos` settings does not violate any rules, and you will not be notified about it, although it may impact the connectivity of your Entity. RxO matching only takes place between DataWriters and DataReaders, the `TopicQos` settings are irrelevant for determining compatibility.

There is one exception to this: the durability service will look only look at the `TopicQos` to see whether it needs to prepare storage facilities for a specific Topic. If the `DurabilityQosPolicy` is not set to `TRANSIENT` or `PERSISTENT` on the topic, then no storage facilities will be prepared for it, regardless of the settings of each individual DataWriter. So when the durability is set to `VOLATILE` on the topic, but a DataWriter specifies `TRANSIENT` durability, then the samples of that DataWriter will not be stored by the durability service. Be careful about that, because you will not be notified about such incompatibilities between Topic and DataWriter. The other way around is not a problem: if the topic specifies a `TRANSIENT` durability, but a DataWriter does not want its samples to be stored by the durability service, then it can specify a `VOLATILE` durability. That is not considered a conflict: in that case the service has prepared storage facilities, but the DataWriter intentionally chooses not to use them.

5.4 Deleting Publishers and DataWriters

Of course, at the end of the application we will need to delete the Publisher and DataWriters before we can delete the DomainParticipant itself. We must also not forget to delete the `DDS_PublisherQos` structure that we allocated on heap, which also includes our Partition string sequence. The following code releases all the resources allocated in the previous code fragment:

```
66    /* Remove the DataWriters */
67    status = DDS_Publisher_delete_datawriter(chatPublisher,
68      talker);
69    checkStatus(status,
70      "DDS_Publisher_delete_datawriter (talker)");
71
72    status = DDS_Publisher_delete_datawriter(
73      chatPublisher, nameServer);
74    checkStatus(
75      status, "DDS_Publisher_delete_datawriter (nameServer)");
76
77   /* Remove the Publisher. */
78   status = DDS_DomainParticipant_delete_publisher(
79     participant, chatPublisher);
80   checkStatus(status, "DDS_DomainParticipant_delete_publisher");
81
82   /* De-allocate the PublisherQoS holder. */
83   DDS_free(pub_qos);  // Note that DDS_free recursively
84                       // de-allocates all indirections!!
```

This code seems very straightforward, each entity is deleted by the same factory that created it, and the result status is always checked for correctness. Now also take a look at the part where we release the `DDS_PublisherQos`. As you can probably remember, the `DDS_PublisherQos` is a structure that embeds all QoS Policies relevant to the `DDS_Publisher`. One of these policies is the PartitionQosPolicy, that embeds a sequence containing a number of string elements. The normal way to release all these indirections is to de-allocate all elements in the reverse order in which they were allocated, in other words,:

- Release the *ChatRoom* string of the Partition sequence.
- Release the sequence buffer itself.
- Release the `DDS_PublisherQos`.

All these steps are automatically performed by the `DDS_free` function, which is very powerful: its function parameter is un-typed, so it can be used to release any type of memory (including all its indirections) that has been allocated using the specialized DDS allocation functions. In this case it will recursively traverse through all attributes of the `DDS_PublisherQos`, release all encountered indirections in there (provided these have also been allocated by the specialized DDS allocation routines), and then release the `DDS_PublisherQos` itself. So the

specialized DDS allocation and de-allocation routines should always be used in pairs: mixing them up with other allocation algorithms will most definitely result in corruption of your memory.

5.5 Registering Instances and Writing Samples

In this section we will actually write our first samples into the system. The first sample will be of type `Chat_NameService` and will contain our user name and user id. The samples following after that will be our actual chat messages. When we are done and want to leave the Chatroom, we will dispose our user information. For that purpose, the example presented in Section 5.2, *Creating Publishers and DataWriters*, is extended with the following lines of code:

```
85   /* Initialize a data sample for the ChatMessage on heap.
86   Chat_ChatMessage *msg;            // Example on Heap.
87
88   /* Initialize a data sample for the NameServer on stack.
89   Chat_NameService ns;              // Example on Stack.
90   ns.userID = ownID;
91   ns.name = DDS_string_alloc(Chat_MAX_NAME+1);
92   checkHandle(ns.name, "DDS_string_alloc");
93   if (chatterName) {
94     strncpy (ns.name, chatterName, Chat_MAX_NAME + 1);
95  } else {
96     snprintf(ns.name, Chat_MAX_NAME+1, "Chatter %d", ownID);
97  }
98
99  /* Write the user-information into the system
100    (registering the instance implicitly). */
101 status = Chat_NameServiceDataWriter_write(
102   nameServer, &ns, DDS_HANDLE_NIL);
103 checkStatus(status, "Chat_ChatMessageDataWriter_write");
104
105 /* Initialize the chat messages that will be written into
106    the ChatRoom on Heap. */
107 msg = Chat_ChatMessage__alloc();
108 checkHandle(msg, "Chat_ChatMessage__alloc");
109 msg->userID = ownID;
110 msg->index = 0;
111 msg->content = DDS_string_alloc(MAX_MSG_LEN);
112 checkHandle(msg->content, "DDS_string_alloc");
113 if (ownID == TERMINATION_MESSAGE) {
114   snprintf (msg->content, MAX_MSG_LEN, "Termination message.");
115 } else {
116   snprintf(msg->content, MAX_MSG_LEN,
117         "Hi there, I will send you %d more messages.", NUM_MSG);
118 }
119
120 /* Register a chat message for this user
121    (pre-allocating resources for it!!) */
122 DDS_InstanceHandle_t userHandle;
123 userHandle = Chat_ChatMessageDataWriter_register_instance(
124   talker, msg);
125
126 /* Write a message using the pre-generated instance handle. */
127 status = Chat_ChatMessageDataWriter_write(
```

```
128   talker, msg, userHandle);
129 checkStatus(status, "Chat_ChatMessageDataWriter_write");
130
131 sleep (1); /* do not run so fast! */
132
133 /* Write any number of messages, re-using the existing
134    string-buffer: no leak!!. */
135 for (i = 1; i <= NUM_MSG && ownID != TERMINATION_MESSAGE; i++) {
136   msg->index = i;
137   snprintf (msg->content, MAX_MSG_LEN,
138             "Message no. %d", msg->index);
139   status = Chat_ChatMessageDataWriter_write(
140     talker, msg, userHandle);
141   checkStatus(status, "Chat_ChatMessageDataWriter_write");
142   sleep (1); /* do not run so fast! */
143 }
```

We first start with the allocation of two samples for the data types that we will be writing. For demonstrational purposes, one of them will be allocated on heap (the `Chat_ChatMessage`) and one will be allocated on stack (the `Chat_NameService`). The advantage of allocating samples on stack is that when they run out of scope, the memory they occupy is automatically reclaimed. However, when such a sample contains indirections, these will have to be released manually in order to avoid a memory leak (see lines *89-91* for the allocation of the NameService sample and its indirection, and line *160* for the de-allocation of this indirection).

In contrast, the `Chat_ChatMessage` sample that is allocated on heap (together with its indirections in lines *107-111*) must be manually de-allocated before it runs out of scope, but by using the `DDS_free` function for that purpose (as demonstrated in line *162* of Section 5.6, *Unregistering and Disposing of Instances*) all indirections will recursively be released as well.

Every sample we write into the system belongs to a specific instance, which is identified by the values of its keyfields. The identity of the `Chat_NameService` sample is determined by its userID field. The `Chat_NameService` sample we intend to write will effectively introduce a new instance into the system. Normally it is a good habit to announce the creation of a new instance, so that the system can pre-allocate and reserve resources for the samples that are to come. This means that the time it takes to write samples describing the state of that instance (which is often the main loop of your applications) can be minimized, since the administrative overhead has already been incurred outside the main loop. In this specific situation, where we only write one sample in the entire lifetime of the instance, it doesn't really profit to announce the existence of the instance explicitly.

Therefore in line *101* we will just write the sample immediately, using the typed DataWriter function `Chat_NameServiceDataWriter_write`, as it is generated by the OpenSplice preprocessor. Again, the first parameter represents the DataWriter that actually performs the operation, the second parameter must be a

pointer to the sample we intend to write (since it was allocated on stack, we need to use the '&' operator here), and the last parameter is the handle to the instance that corresponds to this sample. Since we did not announce the existence of our instance yet, we have no handle to it and therefore use the special constant `DDS_HANDLE_NIL` instead. This forces the DataWriter to deduce the identity of the sample from its key fields, registering the existence of the instance implicitly during the process.

We are then ready to send our chat messages into the world. Since we intend to write more than one chat message, and each message is only identified by the userID of its sender (which is the same for each message we send), it makes sense to announce our new `Chat_ChatMessage` instance first, so that the Publisher can pre-allocate resources for it and we can get its handle immediately. In line *123* we register the existence of our new instance using the typed DataWriter function `Chat_ChatMessageDataWriter_register_instance`, as it is generated by the OpenSplice preprocessor. Again, the first parameter represents the DataWriter that actually performs the operation and the second parameter must be a pointer to a sample that uniquely identifies the new instance by the values of its keyfields. (Since this time the sample is allocated on heap, we do not need to use the '&' operator here). The result of this operation is a handle that uniquely identifies our instance. We will use it in the subsequent write operations.

Before we start writing the chat messages we will first examine their content to see if one of them resembles a termination message. For our simple chatroom application we need a way to tell the `MessageBoard` that it is allowed to terminate, and we do that by sending a special termination message using our Chatter application. A termination messages is a chat message that has a user ID that resembles the special macro `TERMINATION_MESSAGE`, which is an alias for `-1`. When our Chatter encounters such a message it will write this message to the system and print a special message on the screen stating that it just transmitted a termination message, see lines *113-118*.

When the user ID does not resemble a termination request, we enter a loop in lines *135-143* where we write a number of Chat messages into the system, reusing the same sample over and over again by overwriting its string content. In these consecutive write operations we can now pass the instance handle we obtained as a result of the `register_instance` call, so that the DataWriter does not longer need to process the keyfields of the sample in order to deduce the identity of its corresponding instance.

Be careful with this however: if the identity of the instance, as described by the keyfields of the sample, does not match the handle you supply, you will get undefined behaviour: the DataWriter will not give an error message in such a case[1].

5.6 Unregistering and Disposing of Instances

When an instance is no longer relevant for the system it must be unregistered to be able to release the resources it claimed. An instance not only claims resources on the writer side (for example to accommodate for the re-send buffer in case of reliable transmission) but ultimately also on the reader side (to accommodate for the samples it has received so far). As long as a DataWriter has registered an instance, it indicates to the system that it reserves the right to send future updates of that instance. That means that even the readers will need reserve resources to accommodate for these potential updates. So when a writer drops the intention to update a specific instance any longer, it makes sense to announce this decision to the rest of the system. That way not only the writer itself but also all readers communicating with it may reclaim resources they reserved especially for those potential updates.

Be very vigilant about this: writers that keep adding new instances to the system but that fail to unregister the instances they no longer intend to update will not only drain resources on the writer side but also on all readers connected to this writer. A reader is simply not allowed to cleanup resources for instances that are still registered to a datawriter. Don't be afraid that unregistering an instance on the writer side will immediately clean up its resources on the reader side as well, potentially losing information that the reading application didn't have a chance to consume yet: that is not the case. A reader will only reclaim resources of an instance once the writer has unregistered that instance ***and*** once the reading application has consumed all samples for that instance.

So ultimately each instance introduced by a writer must on some moment in time be unregistered by that writer: it is not relevant whether that instance was registered implicitly or explicitly. Unregistering can be done explicitly by invoking the unregister_instance operation on the appropriate datawriter or implicitly by deleting the datawriter. When the system detects that a datawriter has crashed or has simply been deleted, it will automatically unregister all its instances throughout the system.

Besides unregistering an instance, it is also possible to dispose it. The difference between them is predominantly semantical: an instance that is no longer registered to a DataWriter implies that the system does no longer expect any updates for that instance by that DataWriter. That does not imply anything about the lifecycle of the instance: it could be that the DataWriter crashed or that the DataWriter is no longer able to observe the item whose state it was publishing before. Maybe another

1. Caching the instance handle and passing it to the DataWriter with each sample that you write for that instance saves you some performance, since the DataWriter does not need to extract the identity of the instance from the sample. If the DataWriter was forced to check whether sample and instance handle actually match, you would loose this performance gain.

(backup) DataWriter has also registered the instance and is still able to publish updates for it. In that case a DataReader won't even need to deallocate any resources since it can still expect updates from that other DataWriter for the same instance.

By disposing an instance you explicitly tell the system that the instance is no longer alive, for example because the item whose state you were publishing does no longer exist. Normally that means you no longer expect any updates, so a typical response would be to try to reclaim the resources used by that instance. However, since the dispose does not implicitly release any resources by itself, it is typically followed by an explicit unregister operation. Again, on the DataReader side the resources claimed by a disposed and unregistered instance will only be released **after** the application has consumed all samples for that instance.

```
144 /* Leave room by disposing & unregistering message instance.*/
145 status = Chat_ChatMessageDataWriter_dispose(
146   talker, msg, userHandle);
147 checkStatus(status, "Chat_ChatMessageDataWriter_dispose");
148 status = Chat_ChatMessageDataWriter_unregister_instance(
149   talker, msg, userHandle);
150 checkStatus(
151   status, "Chat_ChatMessageDataWriter_unregister_instance");
152
153 /* Also unregister our name. */
154 status = Chat_NameServiceDataWriter_unregister_instance(
155   nameServer, &ns, DDS_HANDLE_NIL);
156 checkStatus(
157   status, "Chat_NameServiceDataWriter_unregister_instance");
158
159 /* Release the data-samples. */
160 DDS_free(ns.name); // ns allocated on stack:
161                    // explicit de-allocation of indirections!!
162 DDS_free(msg);     // msg allocated on heap:
163                    // implicit de-allocation of indirections!!
```

When we are done writing chat messages in our chatter application, we will dispose and un-register the ChatMessage instance, thus announcing the end of our chat session and freeing the resources that it claimed. For this purpose we will use the typed DataWriter functions `Chat_ChatMessageDataWriter_dispose` and `Chat_ChatMessage DataWriter_unregister_instance`, since they are generated by the OpenSplice preprocessor, in lines *145-149*. Their parameter signature is exactly identical to that of the `Chat_ChatMessageDataWriter_write` operation.

It seems logical to also dispose and unregister our user name from the nameservice after we leave the chatroom, but in this case we want to keep track of our user name for future reference. (For example to prevent others from claiming our unique user ID, or to be able to keep track of a list of favorite chat friends[1].) If we would dispose our user name here, it would be marked for destruction not only in the subscribing chatroom but also in the NameService's transient store, so that late joining subscribers will not be aware of our former existence.

So instead of disposing and unregistering our user name, we only want to unregister it so that it remains available in the transient store. This is more tricky then it looks however, because according to the default QoS settings of a DataWriter, an instance is automatically disposed when it is unregistered. Only ommitting the explicit dispose of a user name will merely result in an implicit dispose upon unregistering of that sane user name. That's why we needed to change the DataWriter's `WriterDataLifecycleQosPolicy` to an `autodispose_unregistered_instances` setting of `FALSE` in lines *47-57* of Section 5.2, *Creating Publishers and DataWriters*.

i Note that in most cases transient data will need to outlive the lifetime of the DataWriter that published it (for example for reasons of fault tolerance), so in general it makes sense to set the `autodispose_unregistered_instances` policy of your transient DataWriters to `FALSE`.

In this particular case, it was not necessary to explicitly unregister the message and the user name instances since both instances will implicitly be unregistered when we delete their datawriters This happens very soon afterwards (see lines *67-73* in the last code example in Section 5.2, *Creating Publishers and DataWriters*). However, in a typical application, the lifetime of an instance is shorter than the lifetime of the DataWriter that publishes it, so it is a good habit to explicitly unregister the instances you no longer need.

This ends the publishing side of our Chatter application. The full code listing of this application is under *Chatter.c* in Appendix A, *C Language Examples' Code*.

1. In fact, the DLRL Tutorial introduces a WhiteList object that contains references to some of the users stored in the NameService's transient store, so that only messages originating from those users will be visible on its WhiteListedMessageBoard. See the DLRL Tutorial for more information on that subject.

CHAPTER

6 Subscribing to Data

In this section, you will be guided to create the first (basic) subscribing part of the chatter application, which is the MessageBoard. You will reuse the ChatMessage topic definition of the previous sections to subscribe to all chat messages and to print each of these messages on the message board, together with the userID of its sender. In a later section we will try to substitute this UserID by the appropriate user name of its sender.

The first section will give a short explanation of the different DDS entities that play a role in the subscribing part of an application and the way in which they interact with the publishing side. The next section will teach you how to create a Subscriber with accompanying DataReaders, and how to delete them afterwards. The last section will show you how to use these DataReaders to access samples, how to obtain information about their life cycles and how to manage the memory that holds these samples.

6.1 Subscribers, DataReaders and their QoS Policies

Subscribers and DataReaders are the building blocks required to retrieve information from your system. Both classes are modelled as Entities, meaning both are controlled by a set of QoS Policies, both have their own StatusCondition, both classes can have their own Listener object attached to them, and both classes can only be created and deleted by means of their corresponding factories. This section will introduce the reasons for separating Subscribers from DataReaders in the DDS specification, present the different objectives of both entities, and explain the way in which they interact with their publishing counterparts.

- ***Subscriber*** - A Subscriber is responsible for collecting information coming from various publications, in other words, the Subscriber decides what information is to be retrieved at what time and in which partition. The QoS Policies of the Subscriber control whether samples will be expected to arrive as coherent sets of information, whether the ordering between them will be preserved, and from which Partitions the information will be retrieved. The DomainParticipant acts as a factory for Subscribers.
- ***DataReader*** - A DataReader is a type specific interface for the Subscriber, in other words, it allows an application to access samples of a specific topic from the Subscriber, which actually collects all incoming samples. A Subscriber acts as a factory for its own set of typed DataReaders, and can subscribe to information that

spans more than one Topic. In such cases, it employs a separate DataReader for each individual Topic. The QoS Policies on each DataReader control for the corresponding data type which of the transmitted samples will be accepted into the Subscriber. This acceptance is allocated on the basis of a Request/Offered (RxO) protocol.

- ***Request/Offered Protocol*** - Some policies are applicable to Topics as well as DataWriters and DataReaders (like durability and reliability for example). We already saw in Section 5.1, *Publishers, DataWriters and their QoS Policies*, that in the cases where there is an overlapping QosPolicy between a Topic and a DataWriter, the DataWriter actually decides how the samples are to be transmitted. The TopicQos is only there to provide the DataWriter with a sensible suggestion, and it is free to make another choice. The DataReader has a similar philosophy: for its QoS Policies that overlap with Topics and DataWriters, the TopicQos only serves as a sensible suggestion and the DataReader is free to make another choice. Although the DataReader cannot control with what policy settings the samples are to be offered by the DataWriters, it can control to which DataWriters it will connect. The Request/Offered protocol specifies that a DataReader will only connect to DataWriters with compatible settings: in other words, when DataReaders do not request "more" than what is offered by the DataWriters[1]. DataWriters will not be able to deliver their samples to DataReaders with incompatible QosPolicy settings[2].
- ***SampleInfo*** - Each sample describes the state of a specific instance and may change the lifecycle of that instance. This lifecycle related information might be of interest to the application and is made available through SampleInfo. Each data sample comes with a corresponding SampleInfo structure that contains, among other things, the following fields:
  - ***SampleState*** - Whether the sample has been read before (`DDS_READ_SAMPLE_STATE`) or not (`DDS_NOT_READ_SAMPLE_STATE`).
  - ***ViewState*** - Whether the corresponding instance has already been observed by the application before (`DDS_NEW_VIEW_STATE`) or not (`DDS_NOT_NEW_VIEW_STATE`).

---

1. The DDS specification explicitly formulates an ordering between the different policy values of each QosPolicy to which the Request/Offered (RxO) protocol applies. For our particular example: the ReliabilityQosPolicy value `RELIABLE` > `BEST_EFFORT` and the DurabilityQosPolicy value `PERSISTENT` > `TRANSIENT` > `TRANSIENT_LOCAL` > `VOLATILE`. Refer to the *OpenSplice DDS C Reference Guide*.
2. If a DataReader and a DataWriter have incompatible QosPolicy settings, then both Entities can be notified of this event by their StatusConditions or by their Listeners: the DataWriter will get an OfferedIncompatibleQosStatus event and the DataReader will get an `RequestedIncompatibleQosStatus` event.

- ***InstanceState*** - Whether the instance is still considered alive (`DDS_ALIVE_INSTANCE_STATE`), has already been disposed (`DDS_NOT_ALIVE_DISPOSED_INSTANCE_STATE`), or is no longer registered in any of the DataWriters that are associated to this DataReader (`DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE`).
- ***SourceTimestamp*** - The time at which the sample was written by the DataWriter.[1]

With these building blocks we should be able to build the first elements of our MessageBoard: an application that collects all chat messages and prints them onto the screen.

6.2 Creating Subscribers and DataReaders

In this section we will start to build our entirely new MessagBoard application. The first steps however, are very similar to the ones we took in our Chatter application and are in fact very common for any type of DDS application:

1. Connect to a Domain.
2. Register the required data types to your DomainParticipant
3. Specify the Topics that you want to use

In the previous section, we implemented the last step by creating two new Topics. Creating a Topic is required when you can not be sure that your Topic definition is already available within your Domain. If it was not, creating the Topic will make its definition available to the Domain. If it already was, then creating the Topic for the second time will have no effect on the Domain: your definition is checked against the already available definition and if it conflicts, your Topic creation fails. If it does not conflict, you just get another proxy to the already existing Topic definition (see also Section 4.5, *Topics as Global Concepts*).

If we already know in advance that the Topic definition that we want to use is already available within our Domain, we can also try to obtain a proxy to it without having to actually recreate the Topic ourselves. We can use the `DDS_DomainParticipant_find_topic` function for that purpose. As always, the first parameter specifies the `DDS_DomainParticipant` object that is to execute our function. The second parameter specifies the name of the Topic for which we want to obtain the proxy, and the third parameter specifies the maximum time we want to wait for the topic definition to become available.[2]

1. For this field is to be interpreted correctly by the DataReader, the time on different nodes within the system should be aligned.
2. It is perfectly possible that the application that actually creates the Topic you are waiting for is started after you have been started. In that case you have to wait until its definition is available.

Be aware that even when you try to find an already existing Topic definition, you still need to register its data type locally within your DomainParticipant to be able to read and write samples of it.

In our particular case, we do not know which application will be started first: the `Chatter` or the `MessageBoard`. In fact, we want to be able to experiment a little bit with this ordering to test the effects of the Durability service. That's why in this case we will just create a similar Topic definition for the `ChatMessage` topic as we did in Section 4.6, *Tailoring QosPolicy Settings*. Since we already know how to do that, we will not repeat all these necessary steps. In the following pieces of code, we will therefore just focus on the parts that have to do with the creation of the subscribing entities.

```
1     DDS_SubscriberQos               *sub_qos;
2     DDS_Subscriber                  chatSubscriber;
3     Chat_ChatMessageDataReader      mbReader;
4     char                            *partitionName = NULL;
5
6     /* Adapt the default SubscriberQos to read from the
7        "ChatRoom" Partition. */
8     partitionName = "ChatRoom";
9     sub_qos = DDS_SubscriberQos__alloc();
10  checkHandle(sub_qos, "DDS_SubscriberQos__alloc");
11  status = DDS_DomainParticipant_get_default_subscriber_qos (
12    participant, sub_qos);
13  checkStatus(
14    status, "DDS_DomainParticipant_get_default_subscriber_qos");
15  sub_qos->partition.name._length = 1;
16  sub_qos->partition.name._maximum = 1;
17  sub_qos->partition.name._buffer = DDS_StringSeq_allocbuf (1);
18  checkHandle(
19    sub_qos->partition.name._buffer, "DDS_StringSeq_allocbuf");
20  sub_qos->partition.name._buffer[0] =
21    DDS_string_alloc (strlen(partitionName));
22  checkHandle(
23    sub_qos->partition.name._buffer[0], "DDS_string_alloc");
24  strcpy (sub_qos->partition.name._buffer[0], partitionName);
25
26  /* Create a Subscriber for the MessageBoard application. */
27  chatSubscriber = DDS_DomainParticipant_create_subscriber(
28    participant, sub_qos, NULL, DDS_ANY_STATUS);
29  checkHandle(
30    chatSubscriber, "DDS_DomainParticipant_create_subscriber");
31
32  /* Create a DataReader for the ChatMessage Topic
33     (using the appropriate QoS). */
34  mbReader = DDS_Subscriber_create_datareader(
35    chatSubscriber,
36    chatMessageTopic,
37    DDS_DATAREADER_QOS_USE_TOPIC_QOS,
38    NULL,
39    DDS_ANY_STATUS);
40  checkHandle(mbReader, "DDS_Subscriber_create_datareader");
```

As you can see, this code is very similar to the code used for creating the publishing part of our Chatter application (see Section 5.2, *Creating Publishers and DataWriters*). Since we want to attach to the same Partition as the Chatter application, we first have to adapt the PartitionQosPolicy of our `DDS_SubscriberQos` holder (which is filled with the default settings in line *11*) in a similar way as we did for the `DDS_PublisherQos` in the Chatter application.

We then instruct the `DDS_DomainParticipant` to create a `DDS_Subscriber` (`DDS_DomainParticipant_create_subscriber`), using this `DDS_SubscriberQos` holder (2nd argument) and no `DDS_SubscriberListener` for all status events (3rd and 4th argument).

In line *34*, we invoke the `DDS_Subscriber_create_datareader` function to instruct the `DDS_Subscriber` (1st parameter) to create a typed DataReader for the `ChatMessage` topic (2nd parameter) with QosPolicy values that are copied directly from the corresponding `DDS_TopicQos` (3rd parameter) and no `DDS_DataReaderListener` for all status events (4th and 5th parameter). For the third parameter we used another convenience macro, which has identical functionality as the one explained in Section 5.2, *Creating Publishers and DataWriters*.

Like we already saw in the Chatter application, at the end of the application we will need to delete all these created Entities before we can delete the `DDS_DomainParticipant`. And we must also not forget to delete the `DDS_SubscriberQos` structure that we allocated on heap, which also includes our Partition string sequence. The following code fragment, which is very similar to the one shown in Section 5.2, *Creating Publishers and DataWriters*, releases all the resources allocated in the previous code.

```
41    /* Remove the DataReader */
42    DDS_Subscriber_delete_datareader(chatSubscriber);
43    checkStatus(status, "DDS_Subscriber_delete_datareader");
44
45    /* Remove the Subscriber. */
46    status = DDS_DomainParticipant_delete_subsciber(
47      participant, chatSubscriber);
48   checkStatus(status, "DDS_DomainParticipant_delete_subscriber");
49
50  /* De-allocate the SubscriberQoS holder. */
51  DDS_free(sub_qos);  // Note that DDS_free recursively
52                      // de-allocates all indirections!!
```

6.3 Managing and Reading Samples

In this section we will actually be reading ChatMessage samples from our DataReader and we will print their contents on the standard output. The MessageBoard will be running in a loop, reading all available samples that correspond to live Chatters. The loop is ended when a termination message is

received: that is a chat message whose userID field resembles `TERMINATION_MESSAGE` (a macro for the value `-1`): see line *78*. The code to do all this is presented below.

```
53   DDS_sequence_Chat_ChatMessage *msgSeq =
54     DDS_sequence_Chat_ChatMessage__alloc();
55   checkHandle(msgSeq, "DDS_sequence_Chat_NamedMessage__alloc");
56   DDS_SampleInfoSeq *infoSeq = DDS_SampleInfoSeq__alloc();
57   checkHandle(infoSeq, "DDS_SampleInfoSeq__alloc");
58   DDS_unsigned_long i;
59
60   DDS_boolean terminated = FALSE;
61   while (!terminated) {
62     /* Note: using read does not remove the samples from
63       unregistered instances from the DataReader. This means
64       that the DataRase would use more and more resources.
65       That's why we use take here instead. */
66
67     status = Chat_ChatMessageDataReader_take(
68      mbReader,
69      msgSeq,
70      infoSeq,
71      DDS_LENGTH_UNLIMITED,
72      DDS_ANY_SAMPLE_STATE,
73      DDS_ANY_VIEW_STATE,
74      DDS_ALIVE_INSTANCE_STATE);
75    checkStatus(status, "Chat_NamedMessageDataReader_take");
76    for (i = 0; i < msgSeq->_length; i++) {
77      Chat_ChatMessage *msg = &(msgSeq->_buffer[i]);
78      if (msg->userID == TERMINATION_MESSAGE) {
79        printf("Termination message received: exiting...\n");
80        terminated = TRUE;
81      } else {
82        printf ("%s: %s\n", msg->userName, msg->content);
83      }
84    }
85    status = Chat_ChatMessageDataReader_return_loan(
86      mbReader, msgSeq, infoSeq);
87    checkStatus(
88      status, "Chat_ChatMessageDataReader_return_loan");
89
90    /* Sleep for some amount of time, as not to consume
91       too much CPU cycles. */
92    usleep(100000);
93  }
```

The most important part of this code is located in lines *67-74*, where samples are obtained from the `Chat_ChatMessageDataReader`, using the typed DataReader function `Chat_ChatMessageDataReader_take`, as it is generated by the OpenSplice DDS preprocessor. This function has a number of interesting characteristics:

- It destructively obtains the samples from the DataReader, meaning the samples will no longer be available next time you access the DataReader. There is also an alternative function that is generated by the OpenSplice preprocessor named

`Chat_ChatMessageDataReader_read` that non-destructively obtains the samples, meaning the samples will still be available next time you access the DataReader.

- Both the take and the read functions are non-blocking, meaning they return what is currently available. If nothing is available then no samples are returned and no time is spent waiting for samples to arrive. If you do want to wait until samples are available you will need to use Listeners or WaitSets for that purpose (see also Chapter 7, *Content-Subscription Profile and Listeners* and Chapter 8, *Waiting for Conditions*). To keep this application as simple as possible we will not wait for data to arrive, but will simply take all available samples every 100 milliseconds. In line *92* we use the *`usleep()`* function (imported from unistd.h) to wait between two successive attempts, as not to use to much processing bandwidth.
- Both the take and the read functions have similar signatures in which the following parameters need to be specified:
  - The DataReader whose samples need to be obtained.
  - A sequence that will hold the returned samples.
  - A sequence that will hold the returned SampleInfo.
  - The maximum number of samples that you want to obtain.
  - A SampleState mask for the samples you want to obtain.
  - A ViewState mask for the samples you want to obtain.
  - An InstanceState mask for the samples you want to obtain.

As stated above, both the samples that are to be obtained and their corresponding `DDS_SampleInfo` are returned in sequences that are to be provided by the application as function input parameters. For that reason, both sequences are prepared in advance in lines *53* and *56* using the allocation functions generated by the OpenSplice preprocessor (for each IDL data type `<type>` in module `<module>`, the preprocessor will generate an allocation function called `DDS_sequence_<module>_<type>__alloc`). As you may have noticed in this example, we allocated the sequences on heap, but we did not allocate their internal buffers. That is because the read and take functions are able to perform the allocation of the sequence buffer on account of the application. Both functions have two modes in which they can be operated:

1. The DataReader can *loan* memory to the application (demonstrated above): the sequence buffers are allocated by the DataReader and 'loaned' to the application. If the application does no longer need the samples, it needs to return the 'loan' to the DataReader. Memory that is loaned to the application cannot be used in subsequent read/take function calls.

2. The DataReader can pre-allocate the sequence buffers himself. The DataReader will then just overwrite the allocated memory with the samples that are to be returned. The application itself is responsible for releasing the buffers when no longer required, but the same buffers can be reused in subsequent read/take function calls.

By not pre-allocating the sequence buffers, you indicate the DataReader of the fact that it has to do the allocation on your account. Since we do not know how much samples we may expect, it is hard to give a good estimate for the number of elements that needs to be pre-allocated in your sequence buffer. That's why we make the DataReader responsible for allocating the memory for us: that way it can exactly allocate the number of elements required to return all available samples that match the specified criteria.

The fourth parameter specifies the maximum number of samples you want to obtain as a result of this call. This is very convenient if you pre-allocate your sequence buffer because it can avoid a buffer overflow, or when you can only process a specific number of samples at maximum. In our case neither applies, so we use the special constant `DDS_LENGTH_UNLIMITED` to indicate any number of samples may be returned.

The last three parameters specify the kind of samples that you want to obtain. In Section 6.1, *Subscribers, DataReaders and their QoS Policies*, we saw that every sample had a number of corresponding states (`DDS_SampleState`, `DDS_ViewState` and `DDS_InstanceState`), each of which is represented by a separate bit value. The `read`/`take` functions allow you to specify in a bit mask exactly which states you are interested in: only samples with a state that satisfies the bit mask will be returned to you. For our MessageBoard, the only requirement is that we obtain samples from live Chatters, in other words, messages that have a `DDS_InstanceState` of `DDS_ALIVE_INSTANCE_STATE`. We don't care about the other states, meaning we can raise all bits in their masks. For this purpose the DDS specification provides a special ANY constant for each mask, which has already raised all the relevant bits. In lines *71-74* both the number and the kind of samples we want to obtain are selected.

When the `read`/`take` function returns, the samples and corresponding `DDS_SampleInfo` are available in the sequences we provided. The exact number of returned samples can be found in the _length field of each sequence. In lines *76-84* we iterate through all of the returned samples and print both their userID and their message content. When we do not longer need both sequences we return the so called 'loan' using the typed DataReader function `Chat_ChatMessageDataReader_return_loan`, as it is generated by the OpenSplice preprocessor. This allows the DataReader to reclaim the allocated memory.

To make a good distinction between *loaned* buffers and pre-allocated buffers, we will present the same code below, now using pre-allocated buffers with an estimated maximum number of 100 elements.

```
94   DDS_sequence_Chat_ChatMessage *msgSeq =
95     DDS_sequence_Chat_ChatMessage__alloc();
96   checkHandle(msgSeq, "DDS_sequence_Chat_NamedMessage__alloc");
97   DDS_SampleInfoSeq *infoSeq = DDS_SampleInfoSeq__alloc();
98   checkHandle(infoSeq, "DDS_SampleInfoSeq__alloc");
99   DDS_unsigned_long i;
100
101  msgSeq->_buffer = DDS_sequence_Chat_ChatMessage_allocbuf(100);
102  checkHandle(
103   msgSeq->_buffer, "DDS_sequence_Chat_ChatMessage_allocbuf");
104 infoSeq->_buffer = DDS_ SampleInfoSeq_allocbuf(100);
105 checkHandle(infoSeq->_buffer, "SampleInfoSeq_allocbuf");
106 DDS_sequence_set_release(msgSeq, TRUE);
107 DDS_sequence_set_release(infoSeq, TRUE);
108
109 while (!terminated) {
110   /* Note: using read does not remove the samples from
111      unregistered instances from the DataReader. This means
112      that the DataRase would use more and more resources.
113      That's why we use take here instead. */
114   status = Chat_ChatMessageDataReader_take(
115     mbReader,
116     msgSeq,
117     infoSeq,
118     DDS_LENGTH_UNLIMITED,
119     DDS_ANY_SAMPLE_STATE,
120     DDS_ANY_VIEW_STATE,
121     DDS_ALIVE_INSTANCE_STATE);
122   checkStatus(status, "Chat_NamedMessageDataReader_read");
123   for (i = 0; i < msgSeq->_length; i++) {
124     Chat_ChatMessage *msg = &(msgSeq->_buffer[i]);
125     if (msg->userID == TERMINATION_MESSAGE) {
126       printf("Termination message received: exiting...\n");
127       terminated = TRUE;
128     } else {
129       printf ("%s: %s\n", msg->userName, msg->content);
130     }
131   }
132
133   /* Sleep for some amount of time, as not to consume too
134      much CPU cycles. */
135   usleep(100000);
136 }
137
138 /* Delete the sequences and their contents. */
139 DDS_free(msgSeq);
140 DDS_free(infoSeq);
```

The main differences with the previous code can be found in lines *101-107* where we actually pre-allocate our sequence buffer. As you can see, pre-allocating the buffer requires another generated function named `DDS_sequence_<module>_<type>_allocbuf`, where the parameter specifies the number of elements that need to be allocated. Another function you see here for

the first time is named `DDS_sequence_set_release` and is responsible for setting the release flag of the sequence. (There is also a corresponding function, `DDS_sequence_get_release()`, that returns the value of the release flag).

This release flag is another property of a sequence in C and describes whether the buffer is actually 'owned' by the sequence or not. If it is owned by the sequence, it means the sequence may release the buffer if it is being de-allocated itself (for example by the `DDS_free()` function). However, if the sequence does not own the memory (for example because it just copied an existing pointer instead of all the contents), it may not release that memory when de-allocated by means of the `DDS_free()` function. Since in this example we explicitly allocate buffer space for the sequence, the sequence may consider itself owner of that memory and that's why we need to set the release flag to `TRUE` as well.

If we look at the release flag of a sequence that has 'loaned' a buffer, we will see that its release flag is set to `FALSE`. That means `DDS_free` will not release the buffer when you de-allocate the sequence. You will explicitly need to return this loan before de-allocating the sequence. The read/take functions will not accept sequences that have a release flag set to `FALSE` and that have allocated more than 0 elements, because it will assume it will then be overwriting 'loaned' buffers.

Another difference is the fact that because we now 'own' the buffers ourselves, we do not longer need to return the loan any more: we simply reuse the same buffers over and over again. Notice that we may still use the special `DDS_LENGTH_UNLIMITED` constant to indicate the number of samples we want to obtain, but in this case it represents 100 samples or less, since that is the maximum number of samples that can be stored in the buffers. It is also possible to specify an exact number instead, but that number may not be bigger then the maximum number of samples that the sequences are able to hold. Specifying a bigger number here will result in a return value of `DDS_PRECONDITION_NOT_MET`.

When we exit our loop now, we still own the sequences and their contents, so we should release them manually by using the `DDS_free()` function for that purpose, see lines *139* and *140*.

That concludes our simple MessageBoard for now. In the next section we will expand the MessageBoard to incorporate some smart algorithms to display the username instead of the userID of the sender of a message.

CHAPTER

7 Content-Subscription Profile and Listeners

In this section we will expand the MessageBoard with some code to display the userName instead of the userID for each chat message and to filter out our own messages. Instead of doing all the necessary processing in our application, we will instruct OpenSplice DDS to substitute the userID with a userName by using the principles of aggregation/selection/projection offered by the MultiTopic.

Unfortunately, the MultiTopic is not supported yet in this version of OpenSplice DDS, so we will be simulating its behaviour using a dedicated data type, a ContentFilteredTopic, a private DataReader and DataWriter, a Listener and a QueryCondition.

The first section will introduce the concepts behind the ContentFilteredTopic, the MultiTopic, the ReadCondition and the QueryCondition. The second section shows us how to employ the MultiTopic in our MessageBoard example. The third section will show us how to simulate this MultiTopic, using the above mentioned building blocks, in dedicated code.

7.1 SQL Controlled Building Blocks

This section explains some of the more advanced API building blocks you can use to access only the data you are interested in. These building blocks allow you to use the SQL selection, aggregation, andprojection facilities to express your interest in a greater detail:

- ***ContentFilteredTopic*** - A `ContentFilteredTopic` allows you to filter out samples based on their state. It allows you to specify the `WHERE` clause of an SQL expression, and each sample that does not match the expression will not be inserted into the attached DataReader.
- ***MultiTopic*** - When information coming from several sources needs to be merged into a single (new) data type, so that it is much easier to handle for the application, the `MultiTopic` is a good candidate. It is more advanced than just a `ContentFilteredTopic` and allows advanced features like:
  - *Projection* - Specifies how each original field is projected into the merged data type (the `AS` clause of the SQL expression).

- *Aggregation* - Select the fields and their Topics that need to merged (using the `SELECT` clause of the SQL expression).
- *Selection* - Specify a filter that the merged data type must pass (using the `WHERE` clause of the SQL expression).

- ***ReadCondition*** - A `ReadCondition` allows you to specify your interest (with respect to `SampleState`, `ViewState` and `InstanceState`) by means of bit masks. It will raise a flag when data is available that matches the criteria. When attached to a `WaitSet`, this will trigger the WaitSet. The `ReadCondition` can be passed to a specialized accessor function, that only returns samples that match its criteria.
- ***QueryCondition*** - A `QueryCondition` is more expressive then a `ReadCondition` and also allows you to specify your interest in more detail by adding an SQL `SELECTION` clause. When used in combination with specialized accessor functions, only samples that satisfy the criteria will be returned.

Using these building blocks, we should be able to expand our MessageBoard and to simulate MultiTopics. The coming sections will show how.

7.2 Creating and Using a MultiTopic

If we want to print the `userName` instead of the `userID` for each `ChatMessage`, we require the merged information from two different Topics. The merge criterion is the `userID`, since that is the common keyfield for both Topics. So the easiest thing to do is to create a new data type that aggregates the user name from the `NameService` Topic with the message and index fields of the `ChatMessage` Topic. An IDL expression for such a merged data type can be found below.

```
1    struct NamedMessage {
2      long      userID          // user ID
3      nameType  userName;       // user name
4      long      index;          // message number
5      string    content;        // message body
6    };
7    #pragma keylist NamedMessage userID
```

As you can see, this is the definition for a data type as the MessageBoard application would like to see it: with `userName` and `content` in one structured data type, were the `userID` acts as the keyfield. The next step the application will have to consider is how to map this 'projection type' onto the existing Topics using an SQL expression. Since we want to filter out our own messages on the MessageBoard, but our `MessageBoard` doesn't know by which `userID` these messages are represented, we will use an SQL parameter for that (that parameter can then later be substituted with the correct value, which will be passed as a command line parameter to the `MessageBoard` application):

```
SELECT userID, name AS userName, index, content
FROM Chat_NameService NATURAL JOIN Chat_ChatMessage
WHERE userID <> %0
```

In the above SQL expression you can clearly distinguish the three different aspects of Projection, Aggregation and Selection. The first line specifies which fields will be copied into the merged projection type: if there is an `AS` clause, the projected field will be named accordingly, if there is no `AS` clause, the projected field will have the same name as its original. The second line specifies the source Topics of these fields: since there is more than one source, the several source Topics need to be JOINED together[1]. The third line specifies the conditions that the merged Topics need to satisfy.

Now the only thing the MessageBoard will need to change in order to print a name instead of a userID is the fact that it also needs to obtain a proxy to the `NameService` Topic now (the code will not be shown for that), that it needs to register the projection type, and that it needs to create the `DDS_MultiTopic` according to the above mentioned SQL expression. The DataReader for the `ChatMessage` Topic can then simply be replaced by a similar DataReader for the `DDS_MultiTopic`, as can be seen in the following code.

```
8   Chat_NamedMessageTypeSupport    namedMessageTS;
9   DDS_StringSeq                   *parameterList;
10  Chat_NamedMessageDataReader     mbReader;
11
12  /* Options: MessageBoard [ownID] */
13  /* Messages having owner ownID will be ignored */
14  parameterList = DDS_StringSeq__alloc();
15  checkHandle(parameterList, "DDS_StringSeq__alloc");
16  parameterList->_length = 1;
17  parameterList->_maximum = 1;
18  parameterList->_buffer = DDS_StringSeq_allocbuf(1);
19  checkHandle(parameterList->_buffer, "DDS_StringSeq_allocbuf");
20
21  if (argc > 1) {
22    parameterList->_buffer[0] = DDS_string_alloc(strlen(argv[1]));
23    checkHandle(parameterList->_buffer[0], "DDS_string_alloc");
24    strcpy (parameterList->_buffer[0], argv[1]);
25  }
26  else
27  {
28    parameterList->_buffer[0] = DDS_string_alloc(1);
29    checkHandle(parameterList->_buffer[0], "DDS_string_alloc");
30    strcpy (parameterList->_buffer[0], "0");
31  }
32
33  /* Register the required data type for NamedMessage. */
34  namedMessageTS = Chat_NamedMessageTypeSupport__alloc();
35  checkHandle(
36    namedMessageTS, "Chat_NamedMessageTypeSupport__alloc");
```

1. In case of a name-clash between two joined Topics: it is possible to indicate the source Topic explicitly by prefixing the field name by the Topic name, separated by a dot.

```
37  status = Chat_NamedMessageTypeSupport_register_type(
38    namedMessageTS,
39    participant,
40    namedMessageTypeName);
41  checkStatus(
42    status, "Chat_NamedMessageTypeSupport_register_type");
43
44  /* Create a multitopic that substitutes the userID with
45     its corresponding userName. */
46  namedMessageTopic = DDS_DomainParticipant_create_multitopic(
47    participant,
48    "Chat_NamedMessage",
49    namedMessageTypeName,
50    "SELECT userID, name AS userName, index, content "
51      "FROM Chat_NameService NATURAL JOIN Chat_ChatMessage "
52      "WHERE userID <> %0",
53    parameterList);
54  checkHandle(
55    namedMessageTopic, "DDS_DomainParticipant_create_multitopic");
56
57  /* Create a DataReader for the NamedMessage Topic
58     (using the appropriate QoS). */
59  chatAdmin = DDS_Subscriber_create_datareader(
60    chatSubscriber,
61    namedMessageTopic,
62    DDS_DATAREADER_QOS_USE_TOPIC_QOS,
63    NULL,
64    DDS_ANY_STATUS);
65  checkHandle(chatAdmin, "DDS_Subscriber_create_datareader");
```

In lines *14-31* you see that the SQL parameter variable (representing our own `userID`) is obtained from the command line. The projection data type is registered in lines *34-40*, under *`namedMessageTS`*. This name is then used in lines *46-53*, where the `DDS_DomainParticipant_create_multitopic` function is called to instruct the `DDS_DomainParticipant` (1st parameter) to create a `DDS_MultiTopic` with the name that is specified in the 2nd parameter for the type that is registered under the name specified by the 3rd parameter. The SQL expression is specified in the 4th parameter, and a sequence containing all parameter values (if applicable) is specified in the 5th parameter. SQL parameter values are always specified as strings, since they can refer to variables of different types, depending on the preceding SQL expression.

As you can see in line *59*, creating a DataReader for a `DDS_MultiTopic` is identical to creating a DataReader for a normal `DDS_Topic`: the same function is used. That is possible because the parameter that specifies the Topic is of type `DDS_TopicDescription`, which is the common parent for `DDS_Topics`, as well as for `DDS_MultiTopics` and `DDS_ContentFilteredTopics`.

The last change we need to make of course is to change the print statement to actually display the `userName` instead of the `userID`. We will not show the code for that here, but you can find the full code listing for the `MessageBoard` under *MessageBoard.c* in *Appendix A*.

7.3 Simulating a MultiTopic Using Other Building Blocks

The code presented in the previous section should work according to the DDS specification, but the problem is that this release of OpenSplice DDS does not yet support the `DDS_MultiTopic`. For that reason, and for educational reasons of course, we will simulate the behaviour of the `DDS_MultiTopic` using other building blocks. The idea is that we substitute the `DDS_DomainParticipant_create_multitopic` function with our own function called `DDS_DomainParticipant_create_simulated_multitopic`. This function will do the following things:

1. It will subscribe itself to both the `NameService` and the `ChatMessage` Topics.
2. It will attach the specified Content Filter to the `ChatMessage` Topic
3. It will attach a Listener to the `ChatMessage` DataReader.
4. For each incoming `ChatMessage` it will issue a Query based on its `userID`, to find the corresponding userName in the NameService.
5. It will then manually merge the results into the projection data type.
6. Finally, it will publish this manually created projection type.

The nice thing about this approach is that we can completely hide its functionality to the `MessageBoard`: the code to make the subscriptions and attach the Listener (steps 1 to 3) can be encapsulated in the `create_simulated_multitopic` call, and the manual merge activities for each incoming `ChatMessage` (all the other steps) can be encapsulated in the Listener implementation. We have isolated all this code from the `MessageBoard` and introduced a separate file named `multitopic.c` for it. We already showed you how to make subscriptions, so we will not repeat those steps here, but it is interesting to demonstrate how to create a `DDS_ContentFilteredTopic`, how to implement and attach a Listener interface and how to use `DDS_QueryConditions` to search for information. Those steps will be presented in the following sections. The full implementation for the `multitopic.c` file can be found under *`multitopic.c`* in Appendix A, *C Language Examples' Code.*

7.3.1 Using a ContentFilteredTopic

To avoid unnecessary merging of information, it makes sense to assure that the newly arriving samples match the interest of the user first (in other words, the `WHERE` clause of his SQL expression). A `DDS_ContentFilteredTopic` is a very convenient in such cases: it allows you to attach an SQL Filter expression to an existing Topic and to create a normal DataReader for it. This DataReader will then only receive samples that match the filter expression of the `DDS_ContentFilteredTopic`.

To avoid awkward string parsing to extract the `WHERE` clause of our MultiTopic SQL expression, we will cheat a little bit and manually provide a compatible filter expression for our `DDS_ContentFilteredTopic`.

```
66    DDS_Topic                          chatMessageTopic;
67    DDS_ContentFilteredTopic           filteredMessageTopic;
68    Chat_ChatMessageDataReader         chatMessageDR;
69    DDS_Duration_t infiniteTimeOut  = DDS_DURATION_INFINITY;
70
71    /* Lookup the original ChatMessage Topic. */
72    chatMessageTopic = DDS_DomainParticipant_find_topic(
73      participant,
74      "Chat_ChatMessage",
75     &infiniteTimeOut);
76   checkHandle(
77     chatMessageTopic,
78     "DDS_DomainParticipant_find_topic (Chat_ChatMessage)");
79
80   /* Create a ContentFilteredTopic to filter out our
81      own ChatMessages. */
82   filteredMessageTopic =
83     DDS_DomainParticipant_create_contentfilteredtopic(
84       participant,
85       "Chat_FilteredMessage",
86       chatMessageTopic,
87       "userID <> %0",
88       expression_parameters);
89   checkHandle(
90     filteredMessageTopic,
91     "DDS_DomainParticipant_create_contentfilteredtopic");
92
93   /* Create a DataReader for the FilteredMessage Topic
94      (using the appropriate QoS). */
95   chatMessageDR = DDS_Subscriber_create_datareader(
96     multiSub,
97     filteredMessageTopic,
98     DDS_DATAREADER_QOS_USE_TOPIC_QOS,
99     NULL,
100    DDS_ANY_STATUS);
101  checkHandle(
102    chatMessageDR,
103    "DDS_Subscriber_create_datareader (ChatMessage)");
```

Since this code is in a separate file from the `MessageBoard`, it does not have access to all variables it needs, except for the ones that were passed as parameters to our `create_simulated_multitopic` function. One of the first things we need is a proxy to the `ChatMessage` Topic. Of course we can create our own, like we did before, but that would require us to specify the same QoS parameters and stuff. Right now is easier to just look up the Topic by name: we used the `DDS_DomainParticipant_find_topic` call for that in lines *72-75*, which returns a new proxy to an existing `DDS_Topic` that is identified by the name specified in its 2nd parameter. If a Topic identified by that name cannot yet be found in the `DDS_DomainParticipant` specified in the 1st parameter, it will wait for the time specified in its 3rd parameter to become available (in case it is created by

another, connected, DomainParticipant). If after the specified time it is still not available, it returns a `NULL` pointer. The time out value we provided here is based on the special constant `DDS_DURATION_INFINITY`, which indicates it should wait indefinitely for the Topic to become available.

An alternative operation we could have used for this purpose was the `DDS_DomainParticipant_lookup_topicdescription`: here you also look for a topic by name, but only in your own DomainParticipant: if it is not yet available, it will immediately return `NULL`. However, this operation also allows you to get proxies to `DDS_ContentFilteredTopics` and `DDS_MultiTopics` that are available in the specified `DDS_DomainParticipant`. Because this means that the result can be of different types, the return type is of type `DDS_TopicDescription`, the common parent for all kinds of Topics.

In lines *83-88* we actually create the ContentFilteredTopic itself: the 2nd parameter specifies the name with which this `DDS_ContentFilteredTopic` can be identified (though only locally in the `DDS_DomainParticipant` specified in the 1st parameter, since ContentFilteredTopic definitions are not communicated to other participants), the 3rd parameter specifies the `DDS_Topic` it should filter on, the 4th parameter specifies the filter expression (in SQL), and the 5th parameter specifies the optional filter parameters. Although we cheated a little bit with the creation of the filter expression, we can reuse the SQL expression parameters from the MultiTopic as is, since they are only applicable to the filter part.

In lines *95-100* you can see that creating a DataReader for a `DDS_ContentFilteredTopic` is similar to creating a DataReader for a normal `DDS_Topic` or a `DDS_MultiTopics`.

7.3.2 Attaching a Listener

One of the problems of the IDL to C language mapping is that it does not state how to map a callback interface to C. OpenSplice DDS has solved that problem (like most well known DDS implementations have done) by mapping the callback interface onto a structure that contains a function pointer for each of the contained callback methods. As an example, the Listener of the `DDS_DataReader` is mapped to a structure named `DDS_DataReaderListener` that contains seven function pointer attributes: one for each of the seven callback methods. Besides that, it also contains one extra pointer called `listener_data`, that can be used to store any type of data that needs to be available during each callback that the Listener will make.

Since we only want to respond to incoming data, we only need to implement the `on_data_available` callback function: the other functions we will leave blank, as is demonstrated in the following code.

```
104  /* Declaration of the DataReaderListener. */
105  static struct DDS_DataReaderListener *msgListener = NULL;
106
107  struct MsgListenerState {
108    /* Type-specific DDS entities */
109    Chat_ChatMessageDataReader       chatMessageDR;
110    Chat_NameServiceDataReader       nameServiceDR;
111    Chat_NamedMessageDataWriter      namedMessageDW;
112
113   /* Query related stuff */
114   DDS_QueryCondition               nameFinder;
115   DDS_StringSeq                    *nameFinderParams;
116 };
117
118 /* Implementation for callback function "on_data_available". */
119 void on_message_available(
120   void *listener_data, DDS_DataReader reader ) {
121     ……
122 };
123
124 /* Allocate the DataReaderListener interface. */
125 msgListener = DDS_DataReaderListener__alloc();
126 checkHandle(msgListener, "DDS_DataReaderListener__alloc");
127
128 /* Fill the listener_data with pointers to all entities
129    needed by the Listener implementation. */
130 struct MsgListenerState *listener_state =
131   malloc(sizeof(struct MsgListenerState));
132 checkHandle(listener_state, "malloc");
133 listener_state->chatMessageDR = chatMessageDR;
134 listener_state->nameServiceDR = nameServiceDR;
135 listener_state->namedMessageDW = namedMessageDW;
136 listener_state->nameFinder = nameFinder;
137 listener_state->nameFinderParams = nameFinderParams;
138 msgListener->listener_data = listener_state;
139
140 /* Assign the function pointer attributes
141    to their implementation functions. */
142 msgListener.on_data_available =
143   (void (*)(void *, DDS_DataReader)) on_message_available;
144 msgListener.on_requested_deadline_missed = NULL;
145 msgListener.on_requested_incompatible_qos = NULL;
146 msgListener.on_sample_rejected = NULL;
147 msgListener.on_liveliness_changed = NULL;
148 msgListener.on_subscription_match = NULL;
149 msgListener.on_sample_lost = NULL;
```

In line *105*, the `DDS_DataReaderListener` struct is allocated on the heap. Each of the function pointer attributes is then assigned to its corresponding function implementation in lines *142-149*, which in this case only concerns the `on_data_available` function that is implemented in lines *119-121*. (The actual implementation for this function will be presented later on). Please note in line *143* that you will need to cast your function implementation into the proper type, to match the attribute definition of the `DDS_DataReaderListener`.

In this case the `on_data_available` callback will need to access the following Entities: it will need to read a sample from the ChatMessage DataReader, Query for a matching `userName` in the NameService DataReader and write a merged sample using the `namedMessageDataWriter`. To be able to access all these Entities during this listener callback, we created a special structure called `MsgListenerState` containing pointers to each of them: see lines *107-116*. To make this information available during each listener callback, we first have to allocate and assign the contents of this struct (see lines *130-137*) and then assign its pointer to the `listener_data` fields of the `DDS_DataReaderListener`, see line *138*.

As you can see, the first parameter of each callback function in each listener type is always named `listener_data`, and is in fact exactly the `listener_data` field you store in the corresponding listener structure. That way you have full control over what type of information should be available for each individual Listener instance. Be aware however that for the Listener itself the `listener_data` is an opaque type, it doesn't know what it represents. The implementation for the callback function will always need to cast the `listener_data` field to its correct type before it will be able to access its contents.

Apart from the `on_data_available` function, all the other function pointer attributes have no corresponding implementation and are assigned to `NULL`. Be careful with this though: if the DataReader tries to invoke a function using a function pointer that is set to `NULL` you will definitely get a Segmentation Violation. That's why we need to make sure that the DataReader never tries to invoke the functions that we didn't implement. We can do that by specifying a Listener bit mask: in other words, a mask that tells the DataReader for which events it may notify the Listener and for which events it may not. Each event is represented by its own bit in the bit mask, and each of these bits has its own identifier. Selecting the events for which you want to receive a callback is thus simply a matter of chaining their identifiers in the bit mask when attaching the Listener. For the data_available event, this identifier is named `DDS_DATA_AVAILABLE_STATUS`, see also Table 3, *Status Events Overview*, on page 27:

```
150  /* Attach the DataReaderListener to the DataReader,
151     only enabling the data_available event. */
152  status = DDS_DataReader_set_listener(
153    cmReader, msgListener, DDS_DATA_AVAILABLE_STATUS);
154  checkStatus(status, "DDS_DataReader_set_listener");
```

i The `DDS_DATA_AVAILABLE_STATUS` is *event-based*, *not state* based: it does not trigger on the availability of data (as its name may imply), but on incoming samples or events that have not yet been viewed by the application.

7.3.3 Using a QueryCondition

As stated in Section 7.3, *Simulating a MultiTopic Using Other Building Blocks*, when a new `ChatMessage` sample triggers the Listener we will have to perform the following steps:

1. Extract its `userID`.
2. Execute a query to look for the corresponding `userName` in the `NameService` DataReader.
3. Manually merge the results into a projection sample.
4. Publish this sample.

You should already be able to write the code for most of the above mentioned steps except for the query part, which will be the focus of this section. Before executing a query, you will first need to describe what you are looking for. In DDS terms it means you will need to create a `DDS_QueryCondition` first, where your interest is expressed in SQL. The next step is then to execute this query in a DataReader and to obtain all samples that satisfy it. Since every query is dedicated to look for a specific `userID`, you might be tempted to create new queries for every incoming `ChatMessage`. However, creating the `DDS_QueryCondition` objects is rather expensive, and since all queries are very similar (they only differ with respect to the value of the `userID` they are looking for), it makes sense to parameterise our `DDS_QueryCondition` and reuse it over and over again, only changing the value of the parameter when required.

Such an approach can save you a lot of performance, especially when the creation of `DDS_QueryConditions` can be done outside the main loop, so that this main loop can limit itself to executing queries and changing their parameters. Following this approach, our example will create the `DDS_QueryCondition` during the `DDS_DomainParticipant_create_simulated_multitopic` call (outside the main loop), and adjust and execute it during the Listener callback (inside the main loop). Let's focus on the creation of the `DDS_QueryCondition` first.

```
155   DDS_StringSeq *nameFinderParams;
156   const char *nameFinderExpr;
157
158   /* Define the SQL expression (using a parameterized value). */
159   nameFinderExpr = "userID = %0";
160
161   /* Allocate and assign the query parameters. */
162   nameFinderParams = DDS_StringSeq__alloc();
163   checkHandle(nameFinderParams, "DDS_StringSeq__alloc");
164  nameFinderParams->_length = 1;
165  nameFinderParams->_maximum = 1;
166  nameFinderParams->_buffer = DDS_StringSeq_allocbuf (1);
167  checkHandle(
168    nameFinderParams->_buffer, "DDS_StringSeq_allocbuf");
169  nameFinderParams->_buffer[0] = DDS_string_alloc(
170    strlen(expression_parameters->_buffer[0]) );
```

```
171 checkHandle(nameFinderParams->_buffer[0], "DDS_string_alloc");
172 /* Large enough to hold biggest value */
173 strcpy(
174   nameFinderParams->_buffer[0],
175   expression_parameters->_buffer[0] );
176 DDS_sequence_set_release(nameFinderParams, TRUE);
177
178 /* Create a QueryCondition to only read corresponding
179    nameService information by key-value. */
180 nameFinder = DDS_DataReader_create_querycondition(
181   nameServiceDR,
182   DDS_ANY_SAMPLE_STATE,
183   DDS_ANY_VIEW_STATE,
184   DDS_ANY_INSTANCE_STATE,
185   nameFinderExpr,
186   nameFinderParams);
187 checkHandle(
188   nameFinder, "DDS_DataReader_create_querycondition");
```

As you can see, line *159* specifies the SQL expression, which simply states that the `userID` should be equal to the first parameter. (Parameters in SQL are numbered starting with zero, and are prefixed by the **%** character). Lines *162-176* allocate and initialize the sequence that will represent the query parameters (in this case only 1). Here also all parameters, regardless of their type, must be represented as strings. We have allocated enough string space to make sure that it can hold even the biggest value of the `userID`.

The `DDS_QueryCondition` itself is created in lines *180-186*, where the 1st parameter specifies the DataReader that has to execute the query, the 2nd, 3rd and 4th parameters specify the desired lifecycle states, the 5th parameter specifies the SQL expression and the 6th parameter its parameters.

So this query will be used during a Listener callback to look up the name for a given `ChatMessage`. Let's take a look at what happens during that Listener callback, when we have read a `ChatMessage` sample and want to find the corresponding `NameService` entry.

```
189  /* Find the corresponding named message. */
190  struct MsgListenerState *listener_state;
191
192  /* Obtain all entities mentioned in the listener state. */
193  listener_state = (struct MsgListenerState *) listener_data;
194
195   /* Take available samples and process each one individually. */
196  ....
197
198  if (infoSeq1._buffer[i].valid_data)
199  {
200    if (msgSeq._buffer[i].userID != previous)
201    {
202      previous = msgSeq._buffer[i].userID;
203      snprintf(
204        listener_state->nameFinderParams->_buffer[0],
205        15, "%d", previous);
206      status = DDS_QueryCondition_set_query_parameters(
```

```
207         listener_state->nameFinder,
208         listener_state->nameFinderParams);
209       checkStatus(
210         status, "DDS_QueryCondition_set_query_parameters");
211       status = Chat_NameServiceDataReader_read_w_condition(
212         listener_state->nameServiceDR,
213         &nameSeq,
214         &infoSeq2,
215         DDS_LENGTH_UNLIMITED,
216         listener_state->nameFinder);
217       checkStatus(
218         status, "Chat_NameServiceDataReader_read_w_condition");
219
220       /* Extract Name (there should only be one result). */
221       DDS_free(userName);
222       if (status == DDS_RETCODE_NO_DATA)
223       {
224         userName = DDS_string_alloc(40);
225         checkHandle(userName, "DDS_string_alloc");
226        snprintf(userName, 40, "Name not found!! id = %d",previous);
227       }
228       else
229       {
230         userName = DDS_string_alloc(
231           strlen(nameSeq._buffer[0].name));
232         checkHandle(userName, "DDS_string_alloc");
233           strcpy(userName, nameSeq._buffer[0].name);
234       }
235
236       /* Release the name sample again. */
237       status = Chat_NameServiceDataReader_return_loan(
238         nameServiceDR, &nameSeq, &infoSeq2);
239       checkStatus(
240         status,"Chat_NameServiceDataReader_return_loan");
241     }
242 }
```

The first thing that happens during our listener callback is that we cast the `listener_data` field to a `listener_state` structure, to be able to obtain all the Entities we need during the rest of the callback: see lines *190-193*.

Then we take all available samples (which are not displayed here) and iterate through them. For each sample, we check its SampleInfo to see whether its contents are valid (see line *198*). This is necessary since in some cases a sample is only a placeholder for an instance of a state change. This is an example: the case when a writer disposes of an instance while on the reader side when all samples for that instance have already been taken. Since the dispose operation only changes the instance state, but does not actually transmit the sample it received as one of its parameters,[1] the reader side has no sample with which it can add a change in the

1. Use the writedispose operation if you want both the instance to be disposed and the sample to be transmitted.

SampleInfo. In such cases, the reader will insert a dummy sample of which only the keyfields have any meaningful data. The other fields are not initialized and should therefore not be accessed.

Since this can have drastic consequences for the application, it is important that the application is made aware of which samples are real and which samples are not, so that is does not try to access uninitialized fields of a dummy sample. The field named `valid_data` in the SampleInfo contains exactly that information: if it is `TRUE`, then the sample is a real sample for which all fields are initialized properly, if it is `FALSE`, then only its keyfields should be accessed.

When we know that we have a valid sample, we check (line *200*) whether the current `userID` that needs to be resolved is not equal to the previous one. If so, we still have the previous name and need not look for it again. If this is not the case, we need to look it up anyway, and therefore change the expression parameter to the current `userID`: first we translate the decimal `userID` into a string and insert it into element 0 of the parameter sequence (lines *203-205*), then we use the `DDS_QueryCondition_set_query_parameters` operation to tell the `DDS_QueryCondition` is has to accept this new expression parameter sequence.

We then execute the query on our `Chat_NameServiceDataReader` by invoking the specialized `Chat_NameServiceDataReader_read_w_condition` operation, as it is generated by the OpenSplice preprocessor (lines *211-216*). This operation is similar to the normal `read`/`take` methods, and also has a `take` counterpart. The first 4 parameters are identical to the normal read/take methods, and the last parameter specifies the `DDS_ReadCondition` that the samples need to match. Since a `DDS_QueryCondition` is a specialization of a `DDS_ReadCondition`, it can be used here to make the DataReader only return samples that satisfy our query.

The rest of the code is very straightforward, either one sample is returned (there can be at most one sample that matches the query since `userID` is a key field) or none at all. If there is a sample, we will extract its name, cache it (possibly the next sample that needs to be resolved has the same `userID`) and return the loan. We can then copy the resolved `userName` into the projected data type, together with the content of the `ChatMessage`, and write it into the system. Not all that code is presented here, but see *multitopic.c* in *Appendix A* for the full code listing.

CHAPTER

8 Waiting for Conditions

In this example we will be working on another application called UserLoad, that continuously monitors what is going on in the ChatRoom. It keeps track of all users that come and go, and of all the messages they have sent. It will print a message on the screen when users enter and leave the ChatRoom, and for users that leave the room it will also print the number of messages they have sent while the UserLoad program was monitoring.

For the UserLoad program to detect incoming events, we will use several kinds of Condition objects. A Condition object can be configured to raise a flag when a certain predefined situation occurs. Our application will use different types of Conditions to notify of situations where new users join our ChatRoom, where active users leave it, and when it is time for our application to stop monitoring the ChatRoom. These Conditions are all attached to a WaitSet, that will immediately trigger the main application thread when any of these attached Conditions becomes TRUE.

***Section 8.1,** Conditions and WaitSets, introduces the general rationale behind Conditions and WaitSets and explains the purpose of each Condition type.*

***Section 8.2,** Using a ReadCondition, explains how to use ReadConditions to signal a thread on the arrival of new instances.*

***Section 8.3,** Using a StatusCondition, describes the alternative StatusCondition mechanism to detect when a user leaves the ChatRoom example.*

***Section 8.4,** Using a GuardCondition , describes how a GuardCondition can be used to manually trigger a WaitSet for any user defined reason.*

***Section 8.5,** Using a WaitSet, describes how to attach Conditions to a WaitSet, and how to use this WaitSet to be notified of incoming events.*

***Section 8.6,** Processing Expired Transient Data describes how the transient store treats samples for which the originating writers are no longer alive and the impact it has for an application.*

***Section 8.7,** Using the HistoryQosPolicy, shows how the HistoryQosPolicy can be used to keep track of the history of all messages that are received from the various users.*

*The last section, **Section 8.8,** Cleaning Up, explains how to release resources when an application is terminated.*

8.1 Conditions and WaitSets

There are several different types of Condition objects, each one dedicated to detect a certain type of situation. Each `DDS_Condition` has a flag that becomes `TRUE` when a certain situation occurs, and that remains `TRUE` until that situation has elapsed. The value of this flag can be examined at any time by the application by using the `DDS_Condition_get_trigger_value` operation.

By examining the value of this flag, it is possible for an application to use a polling mechanism to detect the occurrence of a certain event. However, polling might be quite expensive and therefore it may be better to use a mechanism that can block a thread until a certain situation occurs. That is were the `DDS_WaitSet` comes in: a WaitSet allows you to attach any number of `DDS_Condition` objects to it, and to block a thread until one or more of these attached condition objects will have a trigger value that is `TRUE`.

OpenSplice DDS offers the following types of `DDS_Condition` objects:

- ***ReadCondition*** - We already introduced the `DDS_ReadCondition` in Section 7.1, *SQL Controlled Building Blocks*. What we did not mention there is that, since it inherits from the `DDS_Condition` class, it also has a trigger value. This trigger value is `TRUE` as long is data is available that matches the selected lifecycle criteria.
- ***QueryCondition*** - We already introduced the `DDS_QueryCondition` in *SQL Controlled Building Blocks* as well. What we did not mention there is that, since it inherits from the `DDS_ReadCondition`, it also has a trigger value. This trigger value is `TRUE` as long as data is available that matches both the selected lifecycle criteria and the SQL expression.
- ***StatusCondition*** - The `DDS_StatusCondition` was already introduced in *SQL Controlled Building Blocks*. We repeat in here that a `DDS_StatusCondition` may be configured to monitor a user defined set of Entity conditions (being reports of contract violations, reports of conflicting QosPolicy settings with related Entities, reports of the availability of data, etc.), and that the flag of the `DDS_StatusCondition` will be raised as long as at least one of these Entity conditions is `TRUE`.
- ***GuardCondition*** - A `DDS_GuardCondition`'s trigger value is under full control of the application, which can manipulate its state by using the `DDS_StatusCondition_set_trigger_value` operation.

For our `UserLoad` application we will use a `DDS_WaitSet` to block the main thread. The the following `DDS_Conditions` will be attached to this `DDS_WaitSet`:

1. A DDS_ReadCondition that is used to trigger on the event of a new user joining the ChatRoom. It will be created by the NameService DataReader and will be set to trigger on any NameService sample that has a SampleState of NOT_READ, a ViewState of NEW and an InstanceState of ALIVE.
2. A DDS_StatusCondition that is used to trigger on the event of an active user leaving the system. Of course we could do this using another DDS_ReadCondition on the NameService DataReader that would trigger on an InstanceState of NOT_ALIVE_DISPOSED, but for educational purposes we will use the DDS_StatusCondition of the ChatMessage DataReader instead. It will trigger when an associated ChatMessage DataWriter leaves the system.
3. A DDS_GuardCondition that is used to trigger the WaitSet when a pre-defined amount of time has passed. This prevents the UserLoad application from running forever.

When one or more of these Conditions raise their flag, they will trigger the WaitSet, which will then unblock the main application thread. This application thread then receives a list of all the Conditions responsible for the trigger and can handle each one of them individually. The following sections will focus on each of these Conditions.

8.2 Using a ReadCondition

In Section 7.3.3, *Using a QueryCondition*, we already saw an example of how to create a DDS_QueryCondition. Creating a DDS_ReadCondition is very similar to this, since it is a generalization of the DDS_QueryCondition: the only difference is that it doesn't have a corresponding SQL expression.

In our application we want to be informed of new Chatters joining our ChatRoom. Since every Chatter publishes his name and ID in a NameService Topic before joining in, and since each userID represents a unique instance within our NameService DataReader, it seems logical that new instances represent new users joining our ChatRoom. (Note that an instance is marked NEW until its first sample has actually been read. That means that a NEW ViewState can never be combined with a READ SampleState.)

So to detect new users joining our ChatRoom, we only need to get triggered on the arrival of new NameService instances. That means we need to configure our NameService ReadCondition to trigger on samples that have a ViewState of NEW and a SampleState of NOT_READ. Since we only want to signal new users that are still logged in (we will ignore the users that have already logged out before we even got the chance to discover their presence), we will configure our InstanceState to ALIVE. That results in the following code fragment.

```
1    /* A ReadCondition that will contain new users only */
```

```
2     newUser = DDS_DataReader_create_readcondition(
3       nameServer,
4       DDS_NOT_READ_SAMPLE_STATE,
5       DDS_NEW_VIEW_STATE,
6       DDS_ALIVE_INSTANCE_STATE);
7     checkHandle(
8       newUser, "DDS_DataReader_create_readcondition (newUser)");
```

As you can see, its very similar to the code presented in *Using a QueryCondition*. The same approach could also be used to detect users that leave our `ChatRoom`: just select an `InstanceState` that is `NOT_ALIVE_DISPOSED`

In the coming section however, we will use an alternative way of detecting when a user leaves the `ChatRoom`.

8.3 Using a StatusCondition

A `DDS_StatusCondition` is available on every `DDS_Entity` object, just by invoking its `DDS_Entity_get_statuscondition` operation. Since a `DDS_DataReader` inherits from `DDS_Entity`, it also has a `DDS_StatusCondition`. As stated before, `StatusConditions` can be used to notify the Entity of certain situations, like a violation to one of its contracts.

We will not treat each and every possible contract in this tutorial, but we will mention one type of contract here, just to explain the mechanism of the `DDS_StatusCondition`.

When a DataWriter connects to a DataReader, it will establish a contract with it to keep it informed about its Liveliness: in other words, the DataWriter will promise to give a sort of heartbeat to the DataReader, so that the DataReader knows whether it can still expect any updates coming from that DataWriter. If a DataWriter crashes or is deleted, this heartbeat stops, which is a violation of the contract, and so the DataReader must be informed about that. It can then (if applicable[1]) change the `InstanceState` of the concerned instances, in other words, the instances that were being transmitted by that DataWriter, from `ALIVE` to `NOT_ALIVE_NO_WRITERS`. This is important information because it could mean that the resources occupied by these concerned instances may be released after some amount of time[2].

1. Decisions on the liveliness aspects and their consequences are under the control of lots of different QoS Policies, the most important ones being the `LivelinessQosPolicy`, the `OwnershipQosPolicy` and the `DeadlineQosPolicy`. Refer to the *OpenSplice DDS C Reference Guide*.
2. That is controlled by the `ReaderDataLifecycleQosPolicy`.

What is interesting for our application is that we can be notified of the fact that a DataWriter loses its Liveliness, meaning a user effectively leaves the ChatRoom. So besides monitoring the `NameService` for the disposal of a specific `userID`, we can also monitor the Liveliness of each `ChatMessage DataWriter` instead. Let's see how that works.

```
9     /* Obtain a StatusCondition that triggers only when a
10       Writer changes Liveliness */
11    leftUser = DDS_DataReader_get_statuscondition(loadAdmin);
12    checkHandle(leftUser, "DDS_DataReader_get_statuscondition");
13    status = DDS_StatusCondition_set_enabled_statuses(
14      leftUser, DDS_LIVELINESS_CHANGED_STATUS);
  checkStatus(status, "DDS_StatusCondition_set_enabled_statuses");
```

In line *11* you see that we use the `DDS_DataReader_get_statuscondition` operation to obtain the `DDS_StatusCondition` of our `ChatMessage DataReader`. (This operation is inherited from the Entity class and is also available as the `DDS_Entity_get_statuscondition` operation.) By default, the StatusCondition is configured to trigger on all Statuses that are relevant to the corresponding Entity. We only want to respond to the event where a connected `ChatMessage` DataWriter loses its Liveliness, so we will configure the StatusCondition only to trigger on that occasion. The StatusCondition uses a bit mask to select the Statuses it has to monitor, so for that reason we need to set a new bit mask, using the `DDS_StatusCondition_set_enabled_statuses` operation. Each Status is identified by a separate bit and has a unique identifier: the Status we need is named `DDS_LIVELINESS_CHANGED_STATUS`, see also `Table 3, Status Events Overview`, on page 27. You can see in lines *13-14* how we use this identifier to set up the new bit mask.

Once the StatusCondition has triggered, it only means that there is a change in Liveliness in one of the connected DataWriters: the `LivelinessStatus` keeps track of the current number of `alive` DataWriters and of the current number of `not_alive` DataWriters. Any change to these numbers will trigger the StatusCondition. So if we get a trigger, we do not know which user is effected by that, and we do not know whether that user just entered (was added to the `alive_count`) or just left (was removed from the `not_alive_count`). There are two special change counters that keep track of the changes to both the `alive_count` and `not_alive_count`, but these treat both additions and removals in the same way: a removal followed by an addition of a DataWriter leads to an `alive_count_change` of 2. Both change counters will be reset each time the `LivelinessStatus` is obtained.

Since we can't distinguish between users entering and leaving the ChatRoom by just studying the `LivelinessStatus`, we will need to keep track of the previous number of `alive` users. That way we can see whether the current number of users is bigger or smaller than the previous number, and so whether a user has actually

entered or left the ChatRoom. To access this `LivelinessStatus`, we use the `DDS_DataReader_get_liveliness_changed_status` operation on the DataReader.

So now we know when a user actually leaves the ChatRoom, but we still don't know which user that was. We could use a complicated algorithm to map the effected DataWriter to a specific user, but because we already know that a user also unregisters its `userID` in the `NameService` when leaving the room, we will just take all `NOT_ALIVE_NO_WRITERS` instances from the `NameService` DataReader instead, which is much easier to do. So each time we get a trigger for a change in Liveliness, we execute the following code:

```
15   /* Some liveliness has changed (either a DataWriter joined
16      or a DataWriter left) */
17   status = DDS_DataReader_get_liveliness_changed_status(
18     loadAdmin, &livChangStatus);
19   if (livChangStatus.active_count < prevCount) {
20     ...
21     /* A user has left the ChatRoom, since a DataWriter lost
22        its liveliness. Take the effected users so they will
23        not appear in the list later on. */
24    status = Chat_NameServiceDataReader_take(
25      nameServer,
26      &nsList,
27      &infoSeq,
28      DDS_LENGTH_UNLIMITED,
29      DDS_ANY_SAMPLE_STATE,
30      DDS_ANY_VIEW_STATE,
31      DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE);
32    checkStatus(status, " Chat_NameServiceDataReader_take ");
33    ...
34    prevCount = livChangStatus.active_count;
35  };
```

Much more happens in the real code, but we will get back on that later in Section 8.7, *Using the HistoryQosPolicy*.

8.4 Using a GuardCondition

We want the `UserLoad` application to run only for 60 seconds, so we could check in every loop whether this time has already elapsed, and if so, terminate the application. However, if the `UserLoad`'s main thread is blocked on a WaitSet, and no incoming events unblock it, the application can not check our timing constraint and could theoretically be stuck in that WaitSet forever is no more events would allow it to unblock first.

This is where the `DDS_GuardCondition` comes in. As stated before, a `DDS_GuardCondition` is simply a `DDS_Condition` whose trigger value is under full control of the application. In this specific example we could add a GuardCondition to all the other Conditions already attached to the WaitSet. If the `DDS_GuardCondition` has an initial trigger value of `FALSE`, it will not influence

the WaitSet in any way. However, as soon as we change its trigger value into TRUE, the WaitSet must trigger and unblock the main thread, regardless of the settings of all other attached Conditions.

What we will do in our application is to spawn a separate thread that sleeps for 60 seconds. As soon as it wakes up, it will set the trigger value of the DDS_GuardCondition to TRUE. First we will show you how the DDS_GuardCondition is created:

```
36    /* Create a bare guard which will be used to close the room */
37    escape = DDS_GuardCondition__alloc();
38    checkHandle(escape, "DDS_GuardCondition__alloc");
39    ……
40    /* Start the sleeper thread. */
41    pthread_t tid;
42    pthread_create (&tid, NULL, delayedEscape, NULL);
```

As you can see, a DDS_GuardCondition has no corresponding factory and must be created by a DDS_GuardCondition__alloc operation (see line *37*). A new thread is spawned in line *42*, which is instructed to invoke the delayedEscape function as soon as it is ready to be executed. The implementation of that function is depicted below:

```
43    void *
44    delayedEscape(
45      void *arg)
46    {
47      DDS_ReturnCode_t status;
48
49      sleep(60);     /* wait for 60 sec. */
50      status = DDS_GuardCondition_set_trigger_value(escape, TRUE);
51      checkStatus(status, "DDS_GuardCondition_set_trigger_value");
52
53     return NULL;
54  }
```

As you can see, the DDS_GuardCondition is actually a very simple object that can be very convenient if you manually want to unblock a WaitSet. But let's first focus some more on the WaitSet itself, which is the subject of the next section.

8.5 Using a WaitSet

In the previous sections we create a number of DDS_Conditions with the intention of attaching them to a DDS_WaitSet, so that the DDS_WaitSet could unblock the main thread in case of any necessary activity. Let's first see how a DDS_WaitSet is created and how all these DDS_Conditions can be attached to it:

```
55    DDS_WaitSet userLoadWS;
56
57    /* Create a waitset and add the ReadConditions */
58    userLoadWS = DDS_WaitSet__alloc();
```

```
59    checkHandle(userLoadWS, "DDS_WaitSet__alloc");
60    status = DDS_WaitSet_attach_condition(userLoadWS, newUser);
61    checkStatus(status, "DDS_WaitSet_attach_condition (newUser)");
62    status = DDS_WaitSet_attach_condition(userLoadWS, leftUser);
63   checkStatus(status, "DDS_WaitSet_attach_condition (leftUser)");
64  status = DDS_WaitSet_attach_condition(userLoadWS, escape);
65  checkStatus(status, "DDS_WaitSet_attach_condition (escape)");
```

As with the `DDS_GuardCondition`, the `DDS_WaitSet` has no corresponding factory and needs to be created using a special `DDS_WaitSet__alloc` operation (see line *58*). Because of this, there is no dependency on any DomainParticipant and so a `DDS_WaitSet` can be used to combine `DDS_Conditions` coming from different DomainParticipants. This makes `WaitSets` extremely useful to build bridges between several Domains, since they allow an application to react on events coming from different origins.

In lines *60-65* you see how each Condition is attached to the WaitSet, simply using the same `DDS_WaitSet_attach_condition` operation for each type of `Condition`. For the WaitSet it doesn't matter what type of Condition is attached, it will only monitor its trigger value.

Now that we have set up our `DDS_WaitSet`, we can block our main application thread until one of the attached `Conditions` actually raises its flag. In such cases, the WaitSet will unblock and return you the Conditions responsible for that. (Note that more than one `Condition` could have caused the `WaitSet` to unblock). Let's look at the following code, where the main thread blocks itself and then handles the triggered `Conditions`.

```
66    DDS_ConditionSeq *  guardList = NULL;
67    DDS_Duration_t      timeout = DDS_DURATION_INFINITY;
68    int                 closed = 0;
69    DDS_unsigned_long   i, j;
70
71    /* Initialize and pre-allocate the GuardList used to obtain
72       the triggered Conditions. */
73    guardList = DDS_ConditionSeq__alloc();
74    checkHandle(guardList, "DDS_ConditionSeq__alloc");
75   guardList->_maximum = 3;
76   guardList->_length = 0;
77   guardList->_buffer = DDS_ConditionSeq_allocbuf(3);
78   checkHandle(guardList->_buffer, "DDS_ConditionSeq_allocbuf");
79
80    while (!closed) {
81      /* Wait until at least one of the Conditions in
82         the waitset triggers. */
83     status = DDS_WaitSet_wait(userLoadWS, guardList, &timeout);
84    checkStatus(status, "DDS_WaitSet_wait");
85
86    /* Walk over all guards to display information */
87    for (i = 0; i < guardList->_length; i++) {
88      guard = guardList->_buffer[i];
89      if (guard == newUser) {
90        ……
91      } else if (guard == leftUser) {
```

```
92            ........
93          } else if (guard == escape) {
94            printf ("UserLoad has terminated.\n");
95            closed = 1;
96          } else {
97            assert(0);
98          };
99        } /* for */
100 } /* while (!closed) */
```

In line *83* you see how the main thread blocks itself on the WaitSet using the `DDS_WaitSet_wait` operation. The purpose of the `guardList` parameter is to pass back a sequence of all Conditions that were responsible for the trigger: since it is an inout type parameter, we can pre-allocate its contents so that the WaitSet doesn't have to allocate new resources in each and every iteration. Since we already know we attached 3 `Conditions` to the `WaitSet`, the `guardList` can never contain more than 3 elements. That's why we pre-allocate the `guardList` with the worst-case number of elements in lines *73-78*, so that we know we can re-use this buffer in all subsequent iterations without ever having to re-allocate to a bigger buffer. The last parameter specifies how long the `WaitSet` should block at maximum: if the specified time has elapsed but no Condition has triggered, the WaitSet will unblock anyway and return a `DDS_RETCODE_TIMEOUT` and will set the length of the Condition sequence to 0. In this case we have supplied the special constant `DDS_DURATION_INFINITY` to indicate that the WaitSet should wait indefinitely until one of its `Conditions` raises its flag (which is no problem since we use our `GuardCondition` to escape it).

Once the `WaitSet` has triggered, we need to handle all the `Conditions` that were responsible for that. We will do that by just iterating through the `guardList` we obtained (line *87-99*) and comparing each element inside it to the `Conditions` we attached to this `WaitSet`. That way we know which `Condition` represents which purpose, and we can handle each `Condition` in its own special way.

All this said and done, we are almost finished with the `UserLoad` application: the only thing we still need to do is to explain how to keep track of the entire `ChatMessage` history of each `Chatter` that joined our `ChatRoom`. That will be the subject of the next section.

8.6 Processing Expired Transient Data

Since our `UserLoad` application subscribes itself to the `NameService` Topic, which has `TRANSIENT` durability, it will automatically receive all known usernames at startup from the Durability service. Since Chatters leaving the chatroom do not dispose their names from the NameService (see section Section 5.6, *Unregistering and Disposing of Instances*), these names will not be removed from the transient

store[1]. That means that a late joining `NamerService DataReader` at its startup will receive usernames of both currently active users and of users that already left the chatroom.

Since the `UserLoad` application is only interested in displaying usernames of currently active users, it must have a way to filter out the ones that are not currently active. Luckily, the durability service does not influence the instance_state of a sample: if an instance has no active DataWriters, it is delivered with an `instance_state` that is set to `DDS_NOT_ALIVE_NO_WRITERS`, while the ones that do have on or more active DataWriters are set to `DDS_ALIVE`.

```
101  /* Remove all known Users that are not currently active. */
102  status = Chat_NameServiceDataReader_take(
103    nameServer,
104    &nsList,
105    &infoSeq,
106    DDS_LENGTH_UNLIMITED,
107    DDS_ANY_SAMPLE_STATE,
108    DDS_ANY_VIEW_STATE,
109    DDS_NOT_ALIVE_INSTANCE_STATE);
110  checkStatus(status, "Chat_NameServiceDataReader_take");
111  status = Chat_NameServiceDataReader_return_loan(
112    nameServer, &nsList, &infoSeq);
113  checkStatus(status, "Chat_NameServiceDataReader_return_loan");
```

In the listing above, we use of this behaviour to filter out the usernames of user that already left the chatroom. In lines *102-110* we take away all instances that have their instance_state set to `NOT_ALIVE`, leaving only the instances that are currenly still alive[2]. We use the loaning mechanism here because it is difficult to anticipate how many instances that are not considered alive will be delivered by the transient store. That means we may not forget to return the loan right after we took away these samples (see lines *111-113*).

8.7 Using the HistoryQosPolicy

Until now, our DataReaders were configured to store at maximum only one sample for each instance. As you know, a new instance is produced on the DataReader side as soon as its first sample has arrived. When the next sample arrives before the first sample was consumed by the application, it will overwrite the previous one: the idea is that the DataReader always stores the sample that represents the most recent state of an instance.

1. A DataWriter that disposes an instance, also removes it from the transient store. Late joining DataReaders will not be aware of that instance's former existence.
2. The `NOT_ALIVE` instance_state mask specifies both the `NOT_ALIVE_DISPOSED` and `NOT_ALIVE_NO_WRITERS` state.

It may be possible however that you are not interested in just the most recent state of an instance, but that you want to keep track of the latest ***n*** samples of an instance, or maybe even of all samples of an instance. The DataReader can be configured in such a way that it provides you exactly the kind of storage you need. The storage spectrum of a DataReader is under the full control of a QosPolicy named `HistroyQosPolicy`, that has two main settings:

- ***KEEP_LAST*** - This setting comes with a second variable named `depth`. If this depth variable equals `n`, the DataReader will store the latest `n` samples of each instance for you. For newly arriving samples it will behave like a FIFO queue, the oldest sample is shifted out when a new sample arrives.
- ***KEEP_ALL*** - This setting prevents newer samples from overwriting older ones: samples can only disappear when they are actually consumed by the application. If the application does not 'take' its samples and new samples continue to arrive, the DataReader will allocate more and more space, until it reaches its resource limits[1]. If that is the case, it will reject newly arriving samples until the application releases some resources by consuming the older samples. As you may expect, this behaviour can be dangerous if the data is labelled as `RELIABLE`, since the DataWriter may not just drop the data and therefore continuously will need to re-transmit it until it is finally accepted by all the connected DataReaders.

The default HistoryQosPolicy settings are configured to be `KEEP_LAST` with a history depth of ***1***. For our `UserLoad` application we want to keep track of all messages sent by each of the Chatters. That means we will have to change the `HistroyQosPolicy` to `KEEP_ALL`. We do not have a *depth* setting in this case, since the DataReader will just allocate all resources it can claim. This is potentially dangerous if too much users stay online too long, sending out thousands of chat messages while logged in. We will just assume for now that is not the case, so if we take care of the fact that when a user leaves the ChatRoom we will release all the messages it had sent, we should not get into trouble with respect to our resource limits. The following code will show you how to tailor the DataReader QoS settings for this purpose.

```
114   /* Adapt the DataReaderQos for the ChatMessageDataReader
115      to keep track of all messages. */
116   message_qos = DDS_DataReaderQos__alloc();
117   checkHandle(message_qos, "DDS_DataReaderQos__alloc");
118   status = DDS_Subscriber_get_default_datareader_qos(
119     chatSubscriber, message_qos);
120   checkStatus(
121     status, "DDS_Subscriber_get_default_datareader_qos");
122   status = DDS_Subscriber_copy_from_topic_qos(
123     chatSubscriber, message_qos, reliable_topic_qos);
```

1. These resource limits are under full control of the `ResourceLimitsQosPolicy`, and by default are set to unlimited, meaning all the memory available on that specific machine.

```
124   checkStatus(status, "DDS_Subscriber_copy_from_topic_qos");
125   message_qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
126
127  /* Create a DataReader for the ChatMessage Topic (using the
128     appropriate QoS). */
129  loadAdmin = DDS_Subscriber_create_datareader(
130    chatSubscriber,
131    chatMessageTopic,
132    message_qos,
133    NULL,
134    DDS_ANY_STATUS);
135  checkHandle(
136    loadAdmin, "DDS_Subscriber_create_datareader (ChatMessage)");
```

As you can see in line *116*, we start with allocating a `DDS_DataReaderQos` holder, which is filled with the default DataReader settings in line *118*. We then overwrite the QoS Policies that overlap with the Topic policies with the QosPolicy settings of our `ChatMessage` topic in line *122*. Finally we change its History to the `KEEP_ALL` setting in line *125*. (The `HistroyQosPolicy` is no Request/Offered policy, it can be configured independently from the DataWriter settings). We then simply create our DataReader with it in lines *129-134*.

Now let's look at how to access the historical data of a user when he leaves our ChatRoom.

```
137   DDS_LivelinessChangedStatus livChangStatus;
138   DDS_long prevCount = 0;
139
140   if (guard == leftUser) {
141     /* Some liveliness has changed (either a DataWriter
142        joined or a DataWriter left). */
143     status = DDS_DataReader_get_liveliness_changed_status(
144       loadAdmin, &livChangStatus);
145     if (livChangStatus.alive_count < prevCount) {
146       /* A user has left the ChatRoom, since a DataWriter lost
147          its liveliness. Take the effected users so they will not
148          appear in the list later on. */
149      status = Chat_NameServiceDataReader_take(
150        nameServer,
151        &nsList,
152        &infoSeq,
153        DDS_LENGTH_UNLIMITED,
154        DDS_ANY_SAMPLE_STATE,
155        DDS_ANY_VIEW_STATE,
156        DDS_NOT_ALIVE_INSTANCE_STATE);
157      checkStatus(status, "Chat_NameServiceDataReader_take");
158
159      for (j = 0; j < nsList._length; j++) {
160        /* re-apply query arguments */
161        sprintf(args._buffer[0], "%d", nsList._buffer[j].userID);
162        status = DDS_QueryCondition_set_query_parameters(
163          singleUser, &args);
164        checkStatus(
165          status, "DDS_QueryCondition_set_query_parameters");
166
167        /* Read this users history */
168        status = Chat_ChatMessageDataReader_take_w_condition(
```

```
169           loadAdmin,
170           &msgList,
171           &infoSeq2,
172           DDS_LENGTH_UNLIMITED,
173           singleUser);
174         checkStatus(
175           status, "Chat_ChatMessageDataReader_take_w_condition");
176
177         /* Display the user and his history */
178         printf (
179           "Departed user %s has sent %d messages\n",
180           nsList._buffer[j].name,
181           msgList._length);
182         status = Chat_ChatMessageDataReader_return_loan(
183           loadAdmin, &msgList, &infoSeq2);
184         checkStatus(
185           status, "Chat_ChatMessageDataReader_return_loan");
186       }
187       status = Chat_NameServiceDataReader_return_loan(
188         nameServer, &nsList, &infoSeq);
189       checkStatus(
190         status, "Chat_NameServiceDataReader_return_loan");
191     }
192     prevCount = livChangStatus.alive_count;
193 }
```

Parts of this code were already presented in Section 8.3, *Using a StatusCondition*, where we explained how to interpret the StatusCondition trigger. After we have obtained all disposed users by using the instance state `DDS_NOT_ALIVE_INSTANCE_STATE` in lines *149-156*, we iterate over each of these users and try to find their corresponding `ChatMessages` by tailoring the SQL expression parameters (lines *161-165*) and executing this `DDS_QueryCondition` on the `ChatMessage` DataReader (lines *168-173*). We take the data to avoid the DataReader from exhausting its resources, and also because we no longer need `ChatMessages` of a user that has already left. The result is a sequence that contains all the `ChatMessages` of a single user (we queried on `userID`, which is unique for every user), so the length of this sequence tells us how many messages were received from that user.

In case the result would contain information from multiple instances, all samples would still be returned in the same, one dimensional sequence. The ordering of the different samples belonging to the different instances is under full control of the `PresentationQosPolicy` of the Subscriber. When using default policy settings the samples will be ordered as a list, where samples belonging to the same instance are consecutive. The `DDS_SampleInfo` that comes with each sample will give you a *`sample_rank`*, that tells you how much more of the following samples belong to the same instance as the current sample. This may be a very convenient feature if you want to collect all samples that belong to the same instance.

8.8 Cleaning Up

When the `GuardCondition` has triggered our `WaitSet`, and the application leaves its main loop, we need to clean up lots of resources. However, since lots of Entities are currently attached to each other, we will first have to break them apart before we can start to delete them. (Otherwise we would create dangling relationships, in which one entity points to another, already deleted, entity). Let's see what happens when our application leaves the main loop.

```
194  /* Remove all Conditions from the WaitSet. */
195  status = DDS_WaitSet_detach_condition(userLoadWS, escape);
196  checkStatus(status, "DDS_WaitSet_detach_condition (escape)");
197  status = DDS_WaitSet_detach_condition(userLoadWS, leftUser);
198  checkStatus(status, "DDS_WaitSet_detach_condition (leftUser)");
199  status = DDS_WaitSet_detach_condition(userLoadWS, newUser);
200  checkStatus(status, "DDS_WaitSet_detach_condition (newUser)");
201
202  /* Free all resources */
203 DDS_free(guardList);
204 DDS_free(args._buffer);
205 DDS_free(userLoadWS);
206 DDS_free(escape);
207 DDS_free(setting_topic_qos);
208 DDS_free(reliable_topic_qos);
209 DDS_free(nameServiceTypeName);
210 DDS_free(chatMessageTypeName);
211 DDS_free(nameServiceTS);
212 DDS_free(chatMessageTS);
213 status = DDS_DomainParticipant_delete_contained_entities(
214   participant);
215 checkStatus(
216   status, "DDS_DomainParticipant_delete_contained_entities");
217 status = DDS_DomainParticipantFactory_delete_participant(
218   DDS_TheParticipantFactory,
219   participant);
220 checkStatus(
221   status, "DDS_DomainParticipantFactory_delete_participant");
```

In lines *195-200* we detach all Conditions from the `DDS_WaitSet`. Now both the `DDS_WaitSet` and the `DDS_GuardCondition` can be released, and since neither of them has a corresponding factory, we will have to use the `DDS_free` operation to do that (see lines *205*, *206*). After deleting all QoS holders and sequence buffers, it is now time to delete all our Entities using their corresponding factories. Normally we would recursively travel from our `DDS_DomainParticipant` to all its embedded factories, delete all embedded Entities in there, then delete these factories and finally delete our `DDS_DomainParticipant`.

There is however a convenience operation on each factory named `DDS_<factory>_delete_contained_entities` that does exactly that: it recursively travels through all embedded entities and deletes them all. That means that if we invoke it on the `DDS_DomainParticipant` (like we do in line *213*), all

Entities underneath it will be deleted. That leaves only the `DDS_DomainParticipant` itself to be deleted (line *217-219*). Here you see an example of the last convenience macro called `DDS_TheParticipantFactory`.

This macro represents the singleton `DomainParticipantFactory` handle, which can be used at any location where the `DomainParticipantFactory` is required. It allows you to skip the explicit `DDS_DomainParticipantFactory_get_instance` function call that normally provides you with that handle. So creating a `DDS_DomainParticipant` can be as easy as this.

```
222  /* Create a DomainParticipant (using the
223     'TheParticipantFactory' convenience macro). */
224  participant = DDS_DomainParticipantFactory_create_participant(
225    DDS_TheParticipantFactory,
226    domain,
227    DDS_PARTICIPANT_QOS_DEFAULT,
228    NULL,
229    DDS_ANY_STATUS);
```

We do not need to obtain the DomainParticipantFactory handle first; we can just directly insert its convenience macro in here. Be careful when using this specific convenience macro in multi-threaded applications though! Although all other API calls of OpenSplice are re-entrant, the `DDS_DomainParticipantFactory_get_instance` call is not.

Invoking it simultaneously by two or more threads may result in the corruption of memory. This restriction no longer applies after a successful return from its first invocation. Since the convenience macro is just an alias for this function call, it should be used carefully in multi-threaded environments as well.

Finally, as you may have noticed, we did not clean up the sequences used to read and take `NameService` and `ChatMessage` samples. In this case that was not necessary, since we allocated all these sequences on stack.

```
230  DDS_sequence_Chat_ChatMessage msgList =
231    { 0, 0, DDS_OBJECT_NIL, FALSE };
232  DDS_sequence_Chat_NameService nsList =
233    { 0, 0, DDS_OBJECT_NIL, FALSE };
234  DDS_SampleInfoSeq infoSeq = { 0, 0, DDS_OBJECT_NIL, FALSE };
235  DDS_SampleInfoSeq infoSeq2 = { 0, 0, DDS_OBJECT_NIL, FALSE };
```

Allocating a sequence on stack is allowed, but you should not forget to manually release the buffer when the sequence runs out of scope. In this case that was not necessary as well, since we 'loaned' our buffer from the DataReader and we already returned the loan. When allocating sequences on stack though, be sure to initialize them correctly: not only the `_length`, `_maximum` and `_buffer` fields should be initialized correctly, but also the corresponding release flag. According to the IDL C language mapping, this flag can only be set using the appropriate getter and setter

functions (see Section 8.3, *Using a StatusCondition*), but when allocating the sequence on stack it is very convenient to know that the release flag is just a fourth attribute, that can be initialized just like its predecessors.

This completes the tutorial. The full code listing for this application can be found under *UserLoad.c* in *Appendix A*. Of course there is a lot more to learn, especially with regard to all the QoS settings and the corresponding Statuses, but all the basic DDS principles have been covered now. The best way to go from here is to start experimenting yourself now: build some small applications and try to get them to work. While mastering the basics, try to familiarize yourself with the Reference Manual: examine the details of the more complicated API calls and try to get a good overview of all the available QoS settings. Don't be afraid to experiment: it's the best way to increase your knowledge.

Appendices

Appendix

A C Language Examples' Code

This appendix lists the complete C source code for the examples provided in the *C Tutorial Guide.*

Chat.idl

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    Chat.idl
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the data definitions for the tutorial examples.
14  *
15  ***/
16
17 module Chat {
18
19     const long MAX_NAME = 32;
20     typedef string<MAX_NAME> nameType;
21
22     struct ChatMessage {
23         long      userID;          // owner of message
24         long      index;           // message number
25         string    content;         // message body
26     };
27 #pragma keylist ChatMessage userID
28
29     struct NameService {
30         long     userID;           // unique user identification
31         nameType name;             // name of the user
32     };
33 #pragma keylist NameService userID
34
35     struct NamedMessage {
36         long     userID;           // unique user identification
37         nameType userName;         // user name
38         long     index;            // message number
39         string   content;          // message body
40     };
41 #pragma keylist NamedMessage userID
42
43 };
```

CheckStatus.h

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    CheckStatus.h
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the headers for the error handling operations.
14  *
15  ***/
16
17 #ifndef __CHECKSTATUS_H__
18 #define __CHECKSTATUS_H__
19
20 #include "dds_dcps.h"
21 #include <stdio.h>
22 #include <stdlib.h>
23
24 /* Array to hold the names for all ReturnCodes. */
25 char *RetCodeName[13];
26
27 /**
28  * Returns the name of an error code.
29  **/
30 char *getErrorName(DDS_ReturnCode_t status);
31
32 /**
33  * Check the return status for errors. If there is an error, then terminate.
34  **/
35 void checkStatus(DDS_ReturnCode_t status, const char *info);
36
37 /**
38  * Check whether a valid handle has been returned. If not, then terminate.
39  **/
40 void checkHandle(void *handle, char *info);
41
42 #endif
```

CheckStatus.c

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    CheckStatus.c
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C programming language.
```

```
 * DATE              june 2007.
 ************************************************************************
 *
 * This file contains the implementation for the error handling operations.
 *
 ***/

#include "CheckStatus.h"

/* Array to hold the names for all ReturnCodes. */
char *RetCodeName[13] = {
    "DDS_RETCODE_OK",
    "DDS_RETCODE_ERROR",
    "DDS_RETCODE_UNSUPPORTED",
    "DDS_RETCODE_BAD_PARAMETER",
    "DDS_RETCODE_PRECONDITION_NOT_MET",
    "DDS_RETCODE_OUT_OF_RESOURCES",
    "DDS_RETCODE_NOT_ENABLED",
    "DDS_RETCODE_IMMUTABLE_POLICY",
    "DDS_RETCODE_INCONSISTENT_POLICY",
    "DDS_RETCODE_ALREADY_DELETED",
    "DDS_RETCODE_TIMEOUT",
    "DDS_RETCODE_NO_DATA",
    "DDS_RETCODE_ILLEGAL_OPERATION" };

/**
 * Returns the name of an error code.
 **/
char *getErrorName(DDS_ReturnCode_t status)
{
    return RetCodeName[status];
}

/**
 * Check the return status for errors. If there is an error, then terminate.
 **/
void checkStatus(
    DDS_ReturnCode_t status,
    const char *info ) {


    if (status != DDS_RETCODE_OK && status != DDS_RETCODE_NO_DATA) {
        fprintf(stderr, "Error in %s: %s\n", info, getErrorName(status));
        exit (0);
    }
}

/**
 * Check whether a valid handle has been returned. If not, then terminate.
 **/
void checkHandle(
    void *handle,
    char *info ) {

     if (!handle) {
        fprintf(
            stderr,
            "Error in %s: Creation failed: invalid handle\n",
            info);
        exit (0);
     }
```

```
71 }
```

Chatter.c

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    Chatter.c
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the implementation for the 'Chatter' executable.
14  *
15  ***/
16
17 #include <stdlib.h>
18 #include <stdio.h>
19 #include <unistd.h>
20 #include <string.h>
21 #include "dds_dcps.h"
22 #include "CheckStatus.h"
23 #include "Chat.h"
24
25 #define MAX_MSG_LEN 256
26 #define NUM_MSG 10
27 #define TERMINATION_MESSAGE -1
28
29 int
30 main (
31     int argc,
32     char *argv[])
33 {
34     /* Generic DDS entities */
35     DDS_DomainParticipantFactory    dpf;
36     DDS_DomainParticipant           participant;
37     DDS_Topic                       chatMessageTopic;
38     DDS_Topic                       nameServiceTopic;
39     DDS_Publisher                   chatPublisher;
40
41     /* QosPolicy holders */
42     DDS_TopicQos                    *reliable_topic_qos;
43     DDS_TopicQos                    *setting_topic_qos;
44     DDS_PublisherQos                *pub_qos;
45     DDS_DataWriterQos               *dw_qos;
46
47     /* DDS Identifiers */
48     DDS_DomainId_t                  domain = NULL;
49     DDS_InstanceHandle_t            userHandle;
50     DDS_ReturnCode_t                status;
51
52     /* Type-specific DDS entities */
53     Chat_ChatMessageTypeSupport     chatMessageTS;
54     Chat_NameServiceTypeSupport     nameServiceTS;
```

```
55      Chat_ChatMessageDataWriter      talker;
56      Chat_NameServiceDataWriter      nameServer;
57
58      /* Sample definitions */
59      Chat_ChatMessage                *msg;   /* Example on Heap */
60      Chat_NameService                ns;     /* Example on Stack */
61
62      /* Others */
63      int                             ownID = 1;
64      int                             i;
65      char                            *chatMessageTypeName = NULL;
66      char                            *nameServiceTypeName = NULL;
67      char                            *chatterName = NULL;
68      char                            *partitionName = NULL;
69
70
71      /* Options: Chatter [ownID [name]] */
72      if (argc > 1) {
73          sscanf(argv[1], "%d", &ownID);
74          if (argc > 2) {
75              chatterName = argv[2];
76          }
77      }
78
79      /* Create a DomainParticipantFactory and a DomainParticipant
80         (using Default QoS settings). */
81      dpf = DDS_DomainParticipantFactory_get_instance ();
82      checkHandle(dpf, "DDS_DomainParticipantFactory_get_instance");
83      participant = DDS_DomainParticipantFactory_create_participant (
84          dpf,
85          domain,
86          DDS_PARTICIPANT_QOS_DEFAULT,
87          NULL,
88          DDS_ANY_STATUS);
89      checkHandle(
90          participant, "DDS_DomainParticipantFactory_create_participant");
91
92      /* Register the required datatype for ChatMessage. */
93      chatMessageTS = Chat_ChatMessageTypeSupport__alloc();
94      checkHandle(chatMessageTS, "Chat_ChatMessageTypeSupport__alloc");
95      chatMessageTypeName =
96          Chat_ChatMessageTypeSupport_get_type_name(chatMessageTS);
97      status = Chat_ChatMessageTypeSupport_register_type(
98          chatMessageTS,
99          participant,
100         chatMessageTypeName);
101     checkStatus(status, "Chat_ChatMessageTypeSupport_register_type");
102
103     /* Register the required datatype for NameService. */
104     nameServiceTS = Chat_NameServiceTypeSupport__alloc();
105     checkHandle(nameServiceTS, "Chat_NameServiceTypeSupport__alloc");
106     nameServiceTypeName =
107         Chat_NameServiceTypeSupport_get_type_name(nameServiceTS);
108     status = Chat_NameServiceTypeSupport_register_type(
109         nameServiceTS,
110         participant,
111         nameServiceTypeName);
112     checkStatus(status, "Chat_NameServiceTypeSupport_register_type");
113
114     /* Set the ReliabilityQosPolicy to RELIABLE. */
115     reliable_topic_qos = DDS_TopicQos__alloc();
```

```
116     checkHandle(reliable_topic_qos, "DDS_TopicQos__alloc");
117     status = DDS_DomainParticipant_get_default_topic_qos(
118         participant, reliable_topic_qos);
119     checkStatus(status, "DDS_DomainParticipant_get_default_topic_qos");
120     reliable_topic_qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
121
122     /* Make the tailored QoS the new default. */
123     status = DDS_DomainParticipant_set_default_topic_qos(
124         participant, reliable_topic_qos);
125     checkStatus(status, "DDS_DomainParticipant_set_default_topic_qos");
126
127     /* Use the changed policy when defining the ChatMessage topic */
128     chatMessageTopic = DDS_DomainParticipant_create_topic(
129         participant,
130         "Chat_ChatMessage",
131         chatMessageTypeName,
132         reliable_topic_qos,
133         NULL,
134         DDS_ANY_STATUS);
135     checkHandle(
136         chatMessageTopic, "DDS_DomainParticipant_create_topic (ChatMessage)");
137
138     /* Set the DurabilityQosPolicy to TRANSIENT. */
139     setting_topic_qos = DDS_TopicQos__alloc();
140     checkHandle(setting_topic_qos, "DDS_TopicQos__alloc");
141     status = DDS_DomainParticipant_get_default_topic_qos(
142         participant, setting_topic_qos);
143     checkStatus(status, "DDS_DomainParticipant_get_default_topic_qos");
144     setting_topic_qos->durability.kind = DDS_TRANSIENT_DURABILITY_QOS;
145
146     /* Create the NameService Topic. */
147     nameServiceTopic = DDS_DomainParticipant_create_topic(
148         participant,
149         "Chat_NameService",
150         nameServiceTypeName,
151         setting_topic_qos,
152         NULL,
153         DDS_ANY_STATUS);
154     checkHandle(nameServiceTopic, "DDS_DomainParticipant_create_topic");
155
156     /* Adapt the default PublisherQos to write into the
157        "ChatRoom" Partition. */
158     partitionName = "ChatRoom";
159     pub_qos = DDS_PublisherQos__alloc();
160     checkHandle(pub_qos, "DDS_PublisherQos__alloc");
161     status = DDS_DomainParticipant_get_default_publisher_qos(
162         participant, pub_qos);
163     checkStatus(status, "DDS_DomainParticipant_get_default_publisher_qos");
164     pub_qos->partition.name._length = 1;
165     pub_qos->partition.name._maximum = 1;
166     pub_qos->partition.name._buffer = DDS_StringSeq_allocbuf (1);
167     checkHandle(pub_qos->partition.name._buffer, "DDS_StringSeq_allocbuf");
168     pub_qos->partition.name._buffer[0] =
169         DDS_string_alloc( strlen(partitionName) );
170     checkHandle(pub_qos->partition.name._buffer[0], "DDS_string_alloc");
171     strcpy (pub_qos->partition.name._buffer[0], partitionName);
172
173     /* Create a Publisher for the chatter application. */
174     chatPublisher = DDS_DomainParticipant_create_publisher(
175         participant, pub_qos, NULL, DDS_ANY_STATUS);
176     checkHandle(chatPublisher, "DDS_DomainParticipant_create_publisher");
```

```
177
178     /* Create a DataWriter for the ChatMessage Topic
179        (using the appropriate QoS). */
180     talker = DDS_Publisher_create_datawriter(
181         chatPublisher,
182         chatMessageTopic,
183         DDS_DATAWRITER_QOS_USE_TOPIC_QOS,
184         NULL,
185         DDS_ANY_STATUS);
186     checkHandle(talker, "DDS_Publisher_create_datawriter (chatMessage)");
187
188     /* Create a DataWriter for the NameService Topic
189        (using the appropriate QoS). */
190     dw_qos = DDS_DataWriterQos__alloc();
191     checkHandle(dw_qos, "DDS_DataWriterQos__alloc");
192     status = DDS_Publisher_get_default_datawriter_qos (chatPublisher, dw_qos);
193     checkStatus(status, "DDS_Publisher_get_default_datawriter_qos");
194     status = DDS_Publisher_copy_from_topic_qos(
195         chatPublisher, dw_qos, setting_topic_qos);
196     checkStatus(status, "DDS_Publisher_copy_from_topic_qos");
197     dw_qos->writer_data_lifecycle.autodispose_unregistered_instances = FALSE;
198     nameServer = DDS_Publisher_create_datawriter(
199         chatPublisher,
200         nameServiceTopic,
201         dw_qos,
202         NULL,
203         DDS_ANY_STATUS);
204     checkHandle(nameServer, "DDS_Publisher_create_datawriter (NameService)");
205
206     /* Initialize the NameServer attributes located on stack. */
207     ns.userID = ownID;
208     ns.name = DDS_string_alloc(Chat_MAX_NAME+1);
209     checkHandle(ns.name, "DDS_string_alloc");
210     if (chatterName) {
211         strncpy (ns.name, chatterName, Chat_MAX_NAME + 1);
212     } else {
213         snprintf(ns.name, Chat_MAX_NAME+1, "Chatter %d", ownID);
214     }
215
216     /* Write the user-information into the system
217        (registering the instance implicitly). */
218    status = Chat_NameServiceDataWriter_write(nameServer, &ns, DDS_HANDLE_NIL);
219     checkStatus(status, "Chat_ChatMessageDataWriter_write");
220
221     /* Initialize the chat messages on Heap. */
222     msg = Chat_ChatMessage__alloc();
223     checkHandle(msg, "Chat_ChatMessage__alloc");
224     msg->userID = ownID;
225     msg->index = 0;
226     msg->content = DDS_string_alloc(MAX_MSG_LEN);
227     checkHandle(msg->content, "DDS_string_alloc");
228     if (ownID == TERMINATION_MESSAGE) {
229         snprintf (msg->content, MAX_MSG_LEN, "Termination message.");
230     } else {
231         snprintf (msg->content, MAX_MSG_LEN,
232             "Hi there, I will send you %d more messages.", NUM_MSG);
233     }
234     printf("Writing message: %s\n", msg->content);
235
236     /* Register a chat message for this user
237        (pre-allocating resources for it!!) */
```

```
238     userHandle = Chat_ChatMessageDataWriter_register_instance(talker, msg);
239
240     /* Write a message using the pre-generated instance handle. */
241     status = Chat_ChatMessageDataWriter_write(talker, msg, userHandle);
242     checkStatus(status, "Chat_ChatMessageDataWriter_write");
243
244     sleep (1); /* do not run so fast! */
245
246     /* Write any number of messages, re-using the existing
247        string-buffer: no leak!!. */
248     for (i = 1; i <= NUM_MSG && ownID != TERMINATION_MESSAGE; i++) {
249         msg->index = i;
250         snprintf ( msg->content, MAX_MSG_LEN, "Message no. %d", msg->index);
251         printf("Writing message: %s\n", msg->content);
252         status = Chat_ChatMessageDataWriter_write(talker, msg, userHandle);
253         checkStatus(status, "Chat_ChatMessageDataWriter_write");
254         sleep (1); /* do not run so fast! */
255     }
256
257     /* Leave the room by disposing and unregistering the message instance. */
258     status = Chat_ChatMessageDataWriter_dispose(talker, msg, userHandle);
259     checkStatus(status, "Chat_ChatMessageDataWriter_dispose");
260     status = Chat_ChatMessageDataWriter_unregister_instance(
261         talker, msg, userHandle);
262     checkStatus(status, "Chat_ChatMessageDataWriter_unregister_instance");
263
264     /* Also unregister our name. */
265     status = Chat_NameServiceDataWriter_unregister_instance(
266         nameServer, &ns, DDS_HANDLE_NIL);
267     checkStatus(status, "Chat_NameServiceDataWriter_unregister_instance");
268
269     /* Release the data-samples. */
270     DDS_free(ns.name); // ns allocated on stack:
271                        // explicit de-allocation of indirections!!
272     DDS_free(msg);     // msg allocated on heap:
273                        // implicit de-allocation of indirections!!
274
275     /* Remove the DataWriters */
276     status = DDS_Publisher_delete_datawriter(chatPublisher, talker);
277     checkStatus(status, "DDS_Publisher_delete_datawriter (talker)");
278
279     status = DDS_Publisher_delete_datawriter(chatPublisher, nameServer);
280     checkStatus(status, "DDS_Publisher_delete_datawriter (nameServer)");
281
282     /* Remove the Publisher. */
283     status = DDS_DomainParticipant_delete_publisher(
284         participant, chatPublisher);
285     checkStatus(status, "DDS_DomainParticipant_delete_publisher");
286
287     /* Remove the Topics. */
288     status = DDS_DomainParticipant_delete_topic(
289         participant, nameServiceTopic);
290     checkStatus(
291         status, "DDS_DomainParticipant_delete_topic (nameServiceTopic)");
292
293     status = DDS_DomainParticipant_delete_topic(
294         participant, chatMessageTopic);
295     checkStatus(
296         status, "DDS_DomainParticipant_delete_topic (chatMessageTopic)");
297
298     /* De-allocate the QoS policies. */
```

```
299     DDS_free(reliable_topic_qos);
300     DDS_free(setting_topic_qos);
301     DDS_free(pub_qos);  // Note that DDS_free recursively de-allocates
302                         // all indirections as well!!
303
304     /* De-allocate the type-names and TypeSupport objects. */
305     DDS_free(nameServiceTypeName);
306     DDS_free(chatMessageTypeName);
307     DDS_free(nameServiceTS);
308     DDS_free(chatMessageTS);
309
310     /* Remove the DomainParticipant. */
311     status = DDS_DomainParticipantFactory_delete_participant(
312         dpf, participant);
313     checkStatus(status, "DDS_DomainParticipantFactory_delete_participant");
314
315     return 0;
316 }
```

MessageBoard.c

```
1  /*************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    MessageBoard.c
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C programming language.
10  * DATE             june 2007.
11  *************************************************************************
12  *
13  * This file contains the implementation for the 'MessageBoard' executable.
14  *
15  ***/
16
17 #include <stdio.h>
18 #include <unistd.h>
19 #include <string.h>
20
21 #include "dds_dcps.h"
22 #include "CheckStatus.h"
23 #include "Chat.h"
24 #include "multitopic.h"
25
26
27
28 #define TERMINATION_MESSAGE -1
29
30
31
32 int
33 main (
34     int argc,
35     char *argv[])
36 {
37     /* Generic DDS entities */
```

```
38      DDS_DomainParticipantFactory    dpf;
39      DDS_DomainParticipant           participant;
40      DDS_Topic                       chatMessageTopic;
41      DDS_Topic                       nameServiceTopic;
42      DDS_MultiTopic                  namedMessageTopic;
43      DDS_Subscriber                  chatSubscriber;
44
45      /* Type-specific DDS entities */
46      Chat_ChatMessageTypeSupport     chatMessageTS;
47      Chat_NameServiceTypeSupport     nameServiceTS;
48      Chat_NamedMessageTypeSupport    namedMessageTS;
49      Chat_NamedMessageDataReader     chatAdmin;
50      DDS_sequence_Chat_NamedMessage  *msgSeq;
51      DDS_SampleInfoSeq               *infoSeq;
52
53      /* QosPolicy holders */
54      DDS_TopicQos                    *reliable_topic_qos;
55      DDS_TopicQos                    *setting_topic_qos;
56      DDS_SubscriberQos               *sub_qos;
57      DDS_StringSeq                   *parameterList;
58
59      /* DDS Identifiers */
60      DDS_DomainId_t                  domain = NULL;
61      DDS_ReturnCode_t                status;
62
63      /* Others */
64      DDS_unsigned_long               i;
65      DDS_boolean                     terminated = FALSE;
66      char *                          partitionName;
67      char *                          chatMessageTypeName = NULL;
68      char *                          nameServiceTypeName = NULL;
69      char *                          namedMessageTypeName = NULL;
70
71      /* Options: MessageBoard [ownID] */
72      /* Messages having owner ownID will be ignored */
73      parameterList = DDS_StringSeq__alloc();
74      checkHandle(parameterList, "DDS_StringSeq__alloc");
75      parameterList->_length = 1;
76      parameterList->_maximum = 1;
77      parameterList->_buffer = DDS_StringSeq_allocbuf(1);
78      checkHandle(parameterList->_buffer, "DDS_StringSeq_allocbuf");
79
80      if (argc > 1) {
81          parameterList->_buffer[0] = DDS_string_alloc ( strlen(argv[1]) );
82          checkHandle(parameterList->_buffer[0], "DDS_string_alloc");
83          strcpy (parameterList->_buffer[0], argv[1]);
84      }
85      else
86      {
87          parameterList->_buffer[0] = DDS_string_alloc(1);
88          checkHandle(parameterList->_buffer[0], "DDS_string_alloc");
89          strcpy (parameterList->_buffer[0], "0");
90      }
91
92      /* Create a DomainParticipantFactory and a DomainParticipant
93         (using Default QoS settings. */
94      dpf = DDS_DomainParticipantFactory_get_instance ();
95      checkHandle(dpf, "DDS_DomainParticipantFactory_get_instance");
96      participant = DDS_DomainParticipantFactory_create_participant (
97          dpf,
98          domain,
```

```
99          DDS_PARTICIPANT_QOS_DEFAULT,
100         NULL,
101         DDS_ANY_STATUS);
102     checkHandle(
103         participant, "DDS_DomainParticipantFactory_create_participant");
104
105     /* Register the required datatype for ChatMessage. */
106     chatMessageTS = Chat_ChatMessageTypeSupport__alloc();
107     checkHandle(chatMessageTS, "Chat_ChatMessageTypeSupport__alloc");
108     chatMessageTypeName =
109         Chat_ChatMessageTypeSupport_get_type_name(chatMessageTS);
110     status = Chat_ChatMessageTypeSupport_register_type(
111         chatMessageTS,
112         participant,
113         chatMessageTypeName);
114     checkStatus(status, "Chat_ChatMessageTypeSupport_register_type");
115
116     /* Register the required datatype for NameService. */
117     nameServiceTS = Chat_NameServiceTypeSupport__alloc();
118     checkHandle(nameServiceTS, "Chat_NameServiceTypeSupport__alloc");
119     nameServiceTypeName =
120         Chat_NameServiceTypeSupport_get_type_name(nameServiceTS);
121     status = Chat_NameServiceTypeSupport_register_type(
122         nameServiceTS,
123         participant,
124         nameServiceTypeName);
125     checkStatus(status, "Chat_NameServiceTypeSupport_register_type");
126
127     /* Register the required datatype for NamedMessage. */
128     namedMessageTS = Chat_NamedMessageTypeSupport__alloc();
129     checkHandle(namedMessageTS, "Chat_NamedMessageTypeSupport__alloc");
130     namedMessageTypeName =
131         Chat_NamedMessageTypeSupport_get_type_name(namedMessageTS);
132     status = Chat_NamedMessageTypeSupport_register_type(
133         namedMessageTS,
134         participant,
135         namedMessageTypeName);
136     checkStatus(status, "Chat_NamedMessageTypeSupport_register_type");
137
138     /* Set the ReliabilityQosPolicy to RELIABLE. */
139     reliable_topic_qos = DDS_TopicQos__alloc();
140     checkHandle(reliable_topic_qos, "DDS_TopicQos__alloc");
141     status = DDS_DomainParticipant_get_default_topic_qos(
142         participant, reliable_topic_qos);
143     checkStatus(status, "DDS_DomainParticipant_get_default_topic_qos");
144     reliable_topic_qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
145
146     /* Make the tailored QoS the new default. */
147     status = DDS_DomainParticipant_set_default_topic_qos(
148         participant, reliable_topic_qos);
149     checkStatus(status, "DDS_DomainParticipant_set_default_topic_qos");
150
151     /* Use the changed policy when defining the ChatMessage topic */
152     chatMessageTopic = DDS_DomainParticipant_create_topic(
153         participant,
154         "Chat_ChatMessage",
155         chatMessageTypeName,
156         reliable_topic_qos,
157         NULL,
158         DDS_ANY_STATUS);
159     checkHandle(
```

```
160         chatMessageTopic, "DDS_DomainParticipant_create_topic (ChatMessage)");
161
162     /* Set the DurabilityQosPolicy to TRANSIENT. */
163     setting_topic_qos = DDS_TopicQos__alloc();
164     checkHandle(setting_topic_qos, "DDS_TopicQos__alloc");
165     status = DDS_DomainParticipant_get_default_topic_qos(participant,
setting_topic_qos);
166     checkStatus(status, "DDS_DomainParticipant_get_default_topic_qos");
167     setting_topic_qos->durability.kind = DDS_TRANSIENT_DURABILITY_QOS;
168
169     /* Create the NameService Topic. */
170     nameServiceTopic = DDS_DomainParticipant_create_topic(
171         participant,
172         "Chat_NameService",
173         nameServiceTypeName,
174         setting_topic_qos,
175         NULL,
176         DDS_ANY_STATUS);
177     checkHandle(nameServiceTopic, "DDS_DomainParticipant_create_topic");
178
179     /* Create a multitopic that substitutes the userID with
180        its corresponding userName. */
181     namedMessageTopic = DDS_DomainParticipant_create_simulated_multitopic(
182         participant,
183         "Chat_NamedMessage",
184         namedMessageTypeName,
185         "SELECT userID, name AS userName, index, content "
186             "FROM Chat_NameService NATURAL JOIN Chat_ChatMessage "
187             "WHERE userID <> %0",
188         parameterList);
189     checkHandle(
190         namedMessageTopic, "DDS_DomainParticipant_simulate_multitopic");
191
192     /* Adapt the default SubscriberQos to read from the
193        "ChatRoom" Partition. */
194     partitionName = "ChatRoom";
195     sub_qos = DDS_SubscriberQos__alloc();
196     checkHandle(sub_qos, "DDS_SubscriberQos__alloc");
197     status = DDS_DomainParticipant_get_default_subscriber_qos (
198         participant, sub_qos);
199     checkStatus(status, "DDS_DomainParticipant_get_default_subscriber_qos");
200     sub_qos->partition.name._length = 1;
201     sub_qos->partition.name._maximum = 1;
202     sub_qos->partition.name._buffer = DDS_StringSeq_allocbuf (1);
203     checkHandle(sub_qos->partition.name._buffer, "DDS_StringSeq_allocbuf");
204     sub_qos->partition.name._buffer[0] =
205         DDS_string_alloc( strlen(partitionName) );
206     checkHandle(sub_qos->partition.name._buffer[0], "DDS_string_alloc");
207     strcpy (sub_qos->partition.name._buffer[0], partitionName);
208
209     /* Create a Subscriber for the MessageBoard application. */
210     chatSubscriber = DDS_DomainParticipant_create_subscriber(
211         participant, sub_qos, NULL, DDS_ANY_STATUS);
212     checkHandle(chatSubscriber, "DDS_DomainParticipant_create_subscriber");
213
214     /* Create a DataReader for the NamedMessage Topic
215        (using the appropriate QoS). */
216     chatAdmin = DDS_Subscriber_create_datareader(
217         chatSubscriber,
218         namedMessageTopic,
```

```
219         DDS_DATAREADER_QOS_USE_TOPIC_QOS,
220         NULL,
221         DDS_ANY_STATUS);
222     checkHandle(chatAdmin, "DDS_Subscriber_create_datareader");
223
224     /* Print a message that the MessageBoard has opened. */
225     printf( "MessageBoard has opened: send a ChatMessage with "
226                 "userID = -1 to close it....\n\n" );
227
228     /* Allocate the sequence holders for the DataReader */
229     msgSeq = DDS_sequence_Chat_NamedMessage__alloc();
230     checkHandle(msgSeq, "DDS_sequence_Chat_NamedMessage__alloc");
231     infoSeq = DDS_SampleInfoSeq__alloc();
232     checkHandle(infoSeq, "DDS_SampleInfoSeq__alloc");
233
234     while (!terminated) {
235         /* Note: using read does not remove the samples from
236            unregistered instances from the DataReader. This means
237            that the DataRase would use more and more resources.
238            That's why we use take here instead. */
239
240         status = Chat_NamedMessageDataReader_take(
241             chatAdmin,
242             msgSeq,
243             infoSeq,
244             DDS_LENGTH_UNLIMITED,
245             DDS_ANY_SAMPLE_STATE,
246             DDS_ANY_VIEW_STATE,
247             DDS_ALIVE_INSTANCE_STATE );
248         checkStatus(status, "Chat_NamedMessageDataReader_take");
249
250         for (i = 0; i < msgSeq->_length; i++) {
251             Chat_NamedMessage *msg = &(msgSeq->_buffer[i]);
252             if (msg->userID == TERMINATION_MESSAGE) {
253                 printf("Termination message received: exiting...\n");
254                 terminated = TRUE;
255             } else {
256                 printf ("%s: %s\n", msg->userName, msg->content);
257             }
258         }
259
260         status = Chat_NamedMessageDataReader_return_loan(
261             chatAdmin, msgSeq, infoSeq);
262         checkStatus(status, "Chat_ChatMessageDataReader_return_loan");
263
264         /* Sleep for some amount of time, as not to consume
265            too much CPU cycles. */
266         usleep(100000);
267     }
268
269     /* Remove the DataReader */
270     status = DDS_Subscriber_delete_datareader(chatSubscriber, chatAdmin);
271     checkStatus(status, "DDS_Subscriber_delete_datareader");
272
273     /* Remove the Subscriber. */
274     status = DDS_DomainParticipant_delete_subscriber(
275         participant, chatSubscriber);
276     checkStatus(status, "DDS_DomainParticipant_delete_subscriber");
277
278     /* Remove the Topics. */
279     status = DDS_DomainParticipant_delete_simulated_multitopic(
```

```
280         participant, namedMessageTopic);
281     checkStatus(status, "DDS_DomainParticipant_delete_simulated_multitopic");
282
283     status = DDS_DomainParticipant_delete_topic(
284         participant, nameServiceTopic);
285     checkStatus(
286         status, "DDS_DomainParticipant_delete_topic (nameServiceTopic)");
287
288     status = DDS_DomainParticipant_delete_topic(
289         participant, chatMessageTopic);
290     checkStatus(
291         status, "DDS_DomainParticipant_delete_topic (chatMessageTopic)");
292
293     /* De-allocate the QoS policies. */
294     DDS_free(reliable_topic_qos);
295     DDS_free(setting_topic_qos);
296     DDS_free(sub_qos);  // Note that DDS_free recursively de-allocates
297                         // all indirections as well!!
298
299     /* De-allocate the type-names and TypeSupport objects. */
300     DDS_free(namedMessageTypeName);
301     DDS_free(nameServiceTypeName);
302     DDS_free(chatMessageTypeName);
303     DDS_free(namedMessageTS);
304     DDS_free(nameServiceTS);
305     DDS_free(chatMessageTS);
306
307     /* Remove the DomainParticipant. */
308     status = DDS_DomainParticipantFactory_delete_participant(
309         dpf, participant);
310     checkStatus(status, "DDS_DomainParticipantFactory_delete_participant");
311
312     return 0;
313 }
```

multitopic.h

```
1  /*************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    multitopic.h
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C programming language.
10  * DATE             june 2007.
11  *************************************************************************
12  *
13  * This file contains the headers for all operations required to simulate
14  * the MultiTopic behavior.
15  *
16  ***/
17
18 #include "dds_dcps.h"
19
20 DDS_TopicDescription
21 DDS_DomainParticipant_create_simulated_multitopic(
```

```
22      DDS_DomainParticipant participant,
23      const DDS_char *name,
24      const DDS_char *type_name,
25      const DDS_char *subscription_expression,
26      const DDS_StringSeq *expression_parameters
27  );
28
29  DDS_ReturnCode_t
30  DDS_DomainParticipant_delete_simulated_multitopic(
31      DDS_DomainParticipant participant,
32      DDS_TopicDescription smt
33  );
34
35  void on_message_available (
36      void *listener_data,
37      DDS_DataReader reader
38  );
```

multitopic.c

```
1   /*************************************************************************
2    *
3    * Copyright (c) 2007
4    * PrismTech Ltd.
5    * All rights Reserved.
6    *
7    * LOGICAL_NAME:    multitopic.c
8    * FUNCTION:        OpenSplice DDS Tutorial example code.
9    * MODULE:          Tutorial for the C programming language.
10   * DATE             june 2007.
11   *************************************************************************
12   *
13   * This file contains the implementation for all operations required to
14   * simulate the MultiTopic behavior.
15   *
16   ***/
17
18
19  #include <string.h>
20
21  #include "multitopic.h"
22  #include "Chat.h"
23  #include "dds_dcps.h"
24  #include "CheckStatus.h"
25
26  /* DataReaderListener */
27  static struct DDS_DataReaderListener *msgListener = NULL;
28
29  struct MsgListenerState
30  {
31      /* Type-specific DDS entities */
32      Chat_ChatMessageDataReader       chatMessageDR;
33      Chat_NameServiceDataReader       nameServiceDR;
34      Chat_NamedMessageDataWriter      namedMessageDW;
35
36      /* Query related stuff */
37      DDS_QueryCondition               nameFinder;
38      DDS_StringSeq                    *nameFinderParams;
```

```
};

/* Generic DDS entities */
static DDS_Topic                        chatMessageTopic;
static DDS_Topic                        nameServiceTopic;
static DDS_ContentFilteredTopic         filteredMessageTopic;
static DDS_Topic                        namedMessageTopic;
static DDS_Subscriber                   multiSub;
static DDS_Publisher                    multiPub;


DDS_MultiTopic
DDS_DomainParticipant_create_simulated_multitopic (
    DDS_DomainParticipant participant,
    const DDS_char *name,
    const DDS_char *type_name,
    const DDS_char *subscription_expression,
    const DDS_StringSeq *expression_parameters )
{
    /* Type-specific DDS entities */
    static Chat_ChatMessageDataReader       chatMessageDR;
    static Chat_NameServiceDataReader       nameServiceDR;
    static Chat_NamedMessageDataWriter      namedMessageDW;

    /* Query related stuff */
    static DDS_QueryCondition               nameFinder;
    static DDS_StringSeq                    *nameFinderParams;

    /* QosPolicy holders */
    DDS_TopicQos                    *namedMessageQos;
    DDS_SubscriberQos               *sub_qos;
    DDS_PublisherQos                *pub_qos;

    /* Others */
    const char                      *partitionName = "ChatRoom";
    const char                      *nameFinderExpr;
    DDS_Duration_t                  infiniteTimeOut  = DDS_DURATION_INFINITY;
    DDS_ReturnCode_t                status;

    /* Lookup both components that constitute the multi-topic. */
    chatMessageTopic = DDS_DomainParticipant_find_topic(
        participant,
        "Chat_ChatMessage",
        &infiniteTimeOut);
    checkHandle(
        chatMessageTopic,
        "DDS_DomainParticipant_find_topic (Chat_ChatMessage)");

    nameServiceTopic = DDS_DomainParticipant_find_topic(
        participant,
        "Chat_NameService",
        &infiniteTimeOut);
    checkHandle(
        nameServiceTopic,
        "DDS_DomainParticipant_find_topic (Chat_NameService)");

    /* Create a ContentFilteredTopic to filter out our own ChatMessages. */
    filteredMessageTopic = DDS_DomainParticipant_create_contentfilteredtopic(
        participant,
        "Chat_FilteredMessage",
        chatMessageTopic,
```

```
100         "userID <> %0",
101         expression_parameters);
102     checkHandle(
103         filteredMessageTopic,
104         "DDS_DomainParticipant_create_contentfilteredtopic");
105
106
107    /* Adapt the default SubscriberQos to read from the "ChatRoom" Partition. */
108     sub_qos = DDS_SubscriberQos__alloc();
109     checkHandle(sub_qos, "DDS_SubscriberQos__alloc");
110     status = DDS_DomainParticipant_get_default_subscriber_qos(
111         participant, sub_qos);
112     checkStatus(status, "DDS_DomainParticipant_get_default_subscriber_qos");
113     sub_qos->partition.name._length = 1;
114     sub_qos->partition.name._maximum = 1;
115     sub_qos->partition.name._buffer = DDS_StringSeq_allocbuf (1);
116     checkHandle(sub_qos->partition.name._buffer, "DDS_StringSeq_allocbuf");
117     sub_qos->partition.name._buffer[0] =
118         DDS_string_alloc ( strlen(partitionName) );
119     checkHandle(sub_qos->partition.name._buffer[0], "DDS_string_alloc");
120     strcpy (sub_qos->partition.name._buffer[0], partitionName);
121
122     /* Create a private Subscriber for the multitopic simulator. */
123     multiSub = DDS_DomainParticipant_create_subscriber(
124         participant, sub_qos, NULL, DDS_ANY_STATUS);
125     checkHandle(
126         multiSub, "DDS_DomainParticipant_create_subscriber (for multitopic)");
127
128     /* Create a DataReader for the FilteredMessage Topic
129        (using the appropriate QoS). */
130     chatMessageDR = DDS_Subscriber_create_datareader(
131         multiSub,
132         filteredMessageTopic,
133         DDS_DATAREADER_QOS_USE_TOPIC_QOS,
134         NULL,
135         DDS_ANY_STATUS);
136     checkHandle(
137         chatMessageDR, "DDS_Subscriber_create_datareader (ChatMessage)");
138
139     /* Create a DataReader for the nameService Topic
140        (using the appropriate QoS). */
141     nameServiceDR = DDS_Subscriber_create_datareader(
142         multiSub,
143         nameServiceTopic,
144         DDS_DATAREADER_QOS_USE_TOPIC_QOS,
145         NULL,
146         DDS_ANY_STATUS);
147     checkHandle(
148         nameServiceDR, "DDS_Subscriber_create_datareader (NameService)");
149
150     /* Define the SQL expression (using a parameterized value). */
151     nameFinderExpr = "userID = %0";
152
153     /* Allocate and assign the query parameters. */
154     nameFinderParams = DDS_StringSeq__alloc();
155     checkHandle(nameFinderParams, "DDS_StringSeq__alloc");
156     nameFinderParams->_length = 1;
157     nameFinderParams->_maximum = 1;
158     nameFinderParams->_buffer = DDS_StringSeq_allocbuf (1);
159     checkHandle(nameFinderParams->_buffer, "DDS_StringSeq_allocbuf");
160     nameFinderParams->_buffer[0] =
```

```
161         DDS_string_alloc( strlen(expression_parameters->_buffer[0]) );
162     checkHandle(nameFinderParams->_buffer[0], "DDS_string_alloc");
163     strcpy(nameFinderParams->_buffer[0], expression_parameters->_buffer[0]);
164     DDS_sequence_set_release(nameFinderParams, TRUE);
165
166     /* Create a QueryCondition to only read corresponding nameService
167        information by key-value. */
168     nameFinder = DDS_DataReader_create_querycondition(
169         nameServiceDR,
170         DDS_ANY_SAMPLE_STATE,
171         DDS_ANY_VIEW_STATE,
172         DDS_ANY_INSTANCE_STATE,
173         nameFinderExpr,
174         nameFinderParams);
175     checkHandle(
176         nameFinder, "DDS_DataReader_create_querycondition (nameFinder)");
177
178     /* Create the Topic that simulates the multi-topic
179        (use Qos from chatMessage).*/
180     namedMessageQos = DDS_TopicQos__alloc();
181     checkHandle(namedMessageQos, "DDS_TopicQos__alloc");
182     status = DDS_Topic_get_qos(chatMessageTopic, namedMessageQos);
183     checkStatus(status, "DDS_Topic_get_qos");
184
185     /* Create the NamedMessage Topic whose samples simulate the MultiTopic */
186     namedMessageTopic = DDS_DomainParticipant_create_topic(
187         participant,
188         "Chat_NamedMessage",
189         type_name,
190         namedMessageQos,
191         NULL,
192         DDS_ANY_STATUS);
193     checkHandle(
194         namedMessageTopic,
195         "DDS_DomainParticipant_create_topic (NamedMessage)");
196
197     /* Adapt the default PublisherQos to write into the
198        "ChatRoom" Partition. */
199     pub_qos = DDS_PublisherQos__alloc();
200     checkHandle(pub_qos, "DDS_PublisherQos__alloc");
201     status = DDS_DomainParticipant_get_default_publisher_qos (
202         participant, pub_qos);
203     checkStatus(status, "DDS_DomainParticipant_get_default_publisher_qos");
204     pub_qos->partition.name._length = 1;
205     pub_qos->partition.name._maximum = 1;
206     pub_qos->partition.name._buffer = DDS_StringSeq_allocbuf (1);
207     checkHandle(pub_qos->partition.name._buffer, "DDS_StringSeq_allocbuf");
208     pub_qos->partition.name._buffer[0] =
209         DDS_string_alloc( strlen(partitionName) );
210     checkHandle(pub_qos->partition.name._buffer[0], "DDS_string_alloc");
211     strcpy (pub_qos->partition.name._buffer[0], partitionName);
212
213     /* Create a private Publisher for the multitopic simulator. */
214     multiPub = DDS_DomainParticipant_create_publisher(
215         participant, pub_qos, NULL, DDS_ANY_STATUS);
216     checkHandle(
217         multiPub,
218         "DDS_DomainParticipant_create_publisher (for multitopic)");
219
220     /* Create a DataWriter for the multitopic. */
221     namedMessageDW = DDS_Publisher_create_datawriter(
```

```
222         multiPub,
223         namedMessageTopic,
224         DDS_DATAWRITER_QOS_USE_TOPIC_QOS,
225         NULL,
226         DDS_ANY_STATUS);
227     checkHandle(
228         namedMessageDW,
229         "DDS_Publisher_create_datawriter (NamedMessage)");
230 
231     /* Allocate the DataReaderListener interface. */
232     msgListener = DDS_DataReaderListener__alloc();
233     checkHandle(msgListener, "DDS_DataReaderListener__alloc");
234 
235     /* Fill the listener_data with pointers to all entities needed
236        by the Listener implementation. */
237     struct MsgListenerState *listener_state =
238         malloc(sizeof(struct MsgListenerState));
239     checkHandle(listener_state, "malloc");
240     listener_state->chatMessageDR = chatMessageDR;
241     listener_state->nameServiceDR = nameServiceDR;
242     listener_state->namedMessageDW = namedMessageDW;
243     listener_state->nameFinder = nameFinder;
244     listener_state->nameFinderParams = nameFinderParams;
245     msgListener->listener_data = listener_state;
246 
247     /* Assign the function pointer attributes to their
248        implementation functions. */
249     msgListener->on_data_available =
250         (void (*)(void *, DDS_DataReader)) on_message_available;
251     msgListener->on_requested_deadline_missed = NULL;
252     msgListener->on_requested_incompatible_qos = NULL;
253     msgListener->on_sample_rejected = NULL;
254     msgListener->on_liveliness_changed = NULL;
255     msgListener->on_subscription_match = NULL;
256     msgListener->on_sample_lost = NULL;
257 
258     /* Attach the DataReaderListener to the DataReader, only enabling
259        the data_available event. */
260     status = DDS_DataReader_set_listener(
261         chatMessageDR, msgListener, DDS_DATA_AVAILABLE_STATUS);
262     checkStatus(status, "DDS_DataReader_set_listener");
263 
264     /* Free up all resources that are no longer needed. */
265     DDS_free(namedMessageQos);
266     DDS_free(sub_qos);
267     DDS_free(pub_qos);
268 
269     /* Return the simulated Multitopic. */
270     return namedMessageTopic;
271 };
272 
273 DDS_ReturnCode_t
274 DDS_DomainParticipant_delete_simulated_multitopic(
275     DDS_DomainParticipant participant,
276     DDS_TopicDescription smt
277 )
278 {
279     DDS_ReturnCode_t status;
280     struct MsgListenerState *listener_state;
281 
282     /* Obtain all entities mentioned in the listener state. */
```

```
283     listener_state = (struct MsgListenerState *) msgListener->listener_data;
284
285     /* Remove the DataWriter */
286     status = DDS_Publisher_delete_datawriter(
287         multiPub, listener_state->namedMessageDW);
288     checkStatus(status, "DDS_Publisher_delete_datawriter");
289
290     /* Remove the Publisher. */
291     status = DDS_DomainParticipant_delete_publisher(participant, multiPub);
292     checkStatus(status, "DDS_DomainParticipant_delete_publisher");
293
294     /* Remove the QueryCondition and its parameters. */
295     DDS_free(listener_state->nameFinderParams);
296     status = DDS_DataReader_delete_readcondition(
297         listener_state->nameServiceDR,
298         listener_state->nameFinder);
299     checkStatus(status, "DDS_DataReader_delete_readcondition");
300
301     /* Remove the DataReaders. */
302     status = DDS_Subscriber_delete_datareader(
303         multiSub, listener_state->nameServiceDR);
304     checkStatus(status, "DDS_Subscriber_delete_datareader");
305     status = DDS_Subscriber_delete_datareader(
306         multiSub, listener_state->chatMessageDR);
307     checkStatus(status, "DDS_Subscriber_delete_datareader");
308
309     /* Remove the DataReaderListener and its state. */
310     free(listener_state);
311     DDS_free(msgListener);
312
313     /* Remove the Subscriber. */
314     status = DDS_DomainParticipant_delete_subscriber(participant, multiSub);
315     checkStatus(status, "DDS_DomainParticipant_delete_subscriber");
316
317     /* Remove the ContentFilteredTopic. */
318     status = DDS_DomainParticipant_delete_contentfilteredtopic(
319         participant, filteredMessageTopic);
320     checkStatus(status, "DDS_DomainParticipant_delete_contentfilteredtopic");
321
322     /* Remove all other topics. */
323     status = DDS_DomainParticipant_delete_topic(
324         participant, namedMessageTopic);
325     checkStatus(
326         status,
327         "DDS_DomainParticipant_delete_topic (namedMessageTopic)");
328     status = DDS_DomainParticipant_delete_topic(
329         participant, nameServiceTopic);
330     checkStatus(
331         status,
332         "DDS_DomainParticipant_delete_topic (nameServiceTopic)");
333     status = DDS_DomainParticipant_delete_topic(
334         participant,
335         chatMessageTopic);
336     checkStatus(
337         status,
338         "DDS_DomainParticipant_delete_topic (chatMessageTopic)");
339
340     return status;
341 };
342
343
```

```
/* Implementation for the callback function "on_data_available". */
void on_message_available (
    void *listener_data,
    DDS_DataReader reader )
{
    struct MsgListenerState        *listener_state;
    DDS_sequence_Chat_ChatMessage  msgSeq   = { 0, 0, DDS_OBJECT_NIL, FALSE };
    DDS_sequence_Chat_NameService  nameSeq  = { 0, 0, DDS_OBJECT_NIL, FALSE };
    DDS_SampleInfoSeq              infoSeq1 = { 0, 0, DDS_OBJECT_NIL, FALSE };
    DDS_SampleInfoSeq              infoSeq2 = { 0, 0, DDS_OBJECT_NIL, FALSE };
    DDS_ReturnCode_t               status;
    DDS_unsigned_long              i;
    DDS_long                       previous = 0x80000000;
    DDS_string                     userName = DDS_string_alloc(1);


    /* Obtain all entities mentioned in the listener state. */
    listener_state = (struct MsgListenerState *) listener_data;

    /* Take all messages. */
    status = Chat_ChatMessageDataReader_take(
        listener_state->chatMessageDR,
        &msgSeq,
        &infoSeq1,
        DDS_LENGTH_UNLIMITED,
        DDS_ANY_SAMPLE_STATE,
        DDS_ANY_VIEW_STATE,
        DDS_ANY_INSTANCE_STATE);
    checkStatus(status, "Chat_ChatMessageDataReader_take");

    /* For each message, extract the key-field and find
       the corresponding name. */
    for (i = 0; i < msgSeq._length; i++)
    {
        if (infoSeq1._buffer[i].valid_data)
        {
            Chat_NamedMessage joinedSample;

            /* Find the corresponding named message. */
            if (msgSeq._buffer[i].userID != previous)
            {
                previous = msgSeq._buffer[i].userID;
                snprintf(
                    listener_state->nameFinderParams->_buffer[0],
                    15,
                    "%d",
                    previous);
                status = DDS_QueryCondition_set_query_parameters(
                    listener_state->nameFinder,
                    listener_state->nameFinderParams);
               checkStatus(status, "DDS_QueryCondition_set_query_parameters");
                status = Chat_NameServiceDataReader_read_w_condition(
                    listener_state->nameServiceDR,
                    &nameSeq,
                    &infoSeq2,
                    DDS_LENGTH_UNLIMITED,
                    listener_state->nameFinder);
                checkStatus(
                    status, "Chat_NameServiceDataReader_read_w_condition");

                /* Extract Name (there should only be one result). */
```

```
405                 DDS_free(userName);
406                 if (status == DDS_RETCODE_NO_DATA)
407                 {
408                     userName = DDS_string_alloc(40);
409                     checkHandle(userName, "DDS_string_alloc");
410                   snprintf(userName, 40, "Name not found!! id = %d", previous);
411                 }
412                 else
413                 {
414                  userName = DDS_string_alloc(strlen(nameSeq._buffer[0].name));
415                     checkHandle(userName, "DDS_string_alloc");
416                     strcpy(userName, nameSeq._buffer[0].name);
417                 }
418 
419                 /* Release the name sample again. */
420                 status = Chat_NameServiceDataReader_return_loan(
421                     listener_state->nameServiceDR, &nameSeq, &infoSeq2);
422                 checkStatus(status, "Chat_NameServiceDataReader_return_loan");
423             }
424             /* Write merged Topic with both userName and userID. */
425             /* StringCopy not required since sample runs out of
426                scope before string is released. */
427             joinedSample.userName = userName;
428             joinedSample.userID = msgSeq._buffer[i].userID;
429             joinedSample.index = msgSeq._buffer[i].index;
430             joinedSample.content = msgSeq._buffer[i].content;
431                 status = Chat_NamedMessageDataWriter_write(
432                 listener_state->namedMessageDW,
433                 &joinedSample,
434                 DDS_HANDLE_NIL);
435             checkStatus(status, "Chat_NamedMessageDataWriter_write");
436         }
437     }
438     status = Chat_ChatMessageDataReader_return_loan(
439         listener_state->chatMessageDR, &msgSeq, &infoSeq1);
440     checkStatus(status, "Chat_ChatMessageDataReader_return_loan");
441 }
```

UserLoad.c

```
1  /*************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    UserLoad.c
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C programming language.
10  * DATE             june 2007.
11  *************************************************************************
12  *
13  * This file contains the implementation for the 'UserLoad' executable.
14  *
15  ***/
16 
17 #include <stdio.h>
18 #include <unistd.h>
```

```
19 #include <pthread.h>
20 #include <string.h>
21 #include <assert.h>
22
23 #include "dds_dcps.h"
24 #include "CheckStatus.h"
25 #include "Chat.h"
26
27 /* entities required by all threads. */
28 static DDS_GuardCondition           escape;
29
30 /* Sleeper thread: sleeps 60 seconds and then triggers the WaitSet. */
31 void *
32 delayedEscape(
33     void *arg)
34 {
35     DDS_ReturnCode_t status;
36
37     sleep(60);     /* wait for 60 sec. */
38     status = DDS_GuardCondition_set_trigger_value(escape, TRUE);
39     checkStatus(status, "DDS_GuardCondition_set_trigger_value");
40
41     return NULL;
42 }
43
44 int
45 main (
46     int argc,
47     char *argv[])
48 {
49     /* Generic DDS entities */
50     DDS_DomainParticipant           participant;
51     DDS_Topic                       chatMessageTopic;
52     DDS_Topic                       nameServiceTopic;
53     DDS_Subscriber                  chatSubscriber;
54     DDS_QueryCondition              singleUser;
55     DDS_ReadCondition               newUser;
56     DDS_StatusCondition             leftUser;
57     DDS_GuardCondition              guard;
58     DDS_WaitSet                     userLoadWS;
59     DDS_LivelinessChangedStatus     livChangStatus;
60
61     /* QosPolicy holders */
62     DDS_TopicQos                    *setting_topic_qos;
63     DDS_TopicQos                    *reliable_topic_qos;
64     DDS_SubscriberQos               *sub_qos;
65     DDS_DataReaderQos               *message_qos;
66
67     /* DDS Identifiers */
68     DDS_DomainId_t                  domain = NULL;
69     DDS_ReturnCode_t                status;
70     DDS_ConditionSeq                *guardList = NULL;
71     DDS_Duration_t                  timeout = DDS_DURATION_INFINITY;
72
73     /* Type-specific DDS entities */
74     Chat_ChatMessageTypeSupport     chatMessageTS;
75     Chat_NameServiceTypeSupport     nameServiceTS;
76     Chat_NameServiceDataReader      nameServer;
77     Chat_ChatMessageDataReader      loadAdmin;
78     DDS_sequence_Chat_ChatMessage   msgList  = { 0, 0, DDS_OBJECT_NIL, FALSE };
79     DDS_sequence_Chat_NameService   nsList   = { 0, 0, DDS_OBJECT_NIL, FALSE };
```

```
80      DDS_SampleInfoSeq               infoSeq  = { 0, 0, DDS_OBJECT_NIL, FALSE };
81      DDS_SampleInfoSeq               infoSeq2 = { 0, 0, DDS_OBJECT_NIL, FALSE };
82
83      /* Others */
84      DDS_StringSeq                    args;
85      int                              closed = 0;
86      DDS_unsigned_long                i, j;
87      DDS_long                         prevCount = 0;
88      char                             *partitionName;
89      char                             *chatMessageTypeName = NULL;
90      char                             *nameServiceTypeName = NULL;
91      pthread_t                        tid;
92
93      /* Create a DomainParticipant (using the
94         'TheParticipantFactory' convenience macro). */
95      participant = DDS_DomainParticipantFactory_create_participant (
96          DDS_TheParticipantFactory,
97          domain,
98          DDS_PARTICIPANT_QOS_DEFAULT,
99          NULL,
100         DDS_ANY_STATUS);
101     checkHandle(
102         participant, "DDS_DomainParticipantFactory_create_participant");
103
104     /* Register the required datatype for ChatMessage. */
105     chatMessageTS = Chat_ChatMessageTypeSupport__alloc();
106     checkHandle(chatMessageTS, "Chat_ChatMessageTypeSupport__alloc");
107     chatMessageTypeName =
108         Chat_ChatMessageTypeSupport_get_type_name(chatMessageTS);
109     status = Chat_ChatMessageTypeSupport_register_type(
110         chatMessageTS,
111         participant,
112         chatMessageTypeName);
113     checkStatus(status, "Chat_ChatMessageTypeSupport_register_type");
114
115     /* Register the required datatype for NameService. */
116     nameServiceTS = Chat_NameServiceTypeSupport__alloc();
117     checkHandle(nameServiceTS, "Chat_NameServiceTypeSupport__alloc");
118     nameServiceTypeName =
119         Chat_NameServiceTypeSupport_get_type_name(nameServiceTS);
120     status = Chat_NameServiceTypeSupport_register_type(
121         nameServiceTS,
122         participant,
123         nameServiceTypeName);
124     checkStatus(status, "Chat_NameServiceTypeSupport_register_type");
125
126     /* Set the ReliabilityQosPolicy to RELIABLE. */
127     reliable_topic_qos = DDS_TopicQos__alloc();
128     checkHandle(reliable_topic_qos, "DDS_TopicQos__alloc");
129     status = DDS_DomainParticipant_get_default_topic_qos(
130         participant, reliable_topic_qos);
131     checkStatus(status, "DDS_DomainParticipant_get_default_topic_qos");
132     reliable_topic_qos->reliability.kind = DDS_RELIABLE_RELIABILITY_QOS;
133
134     /* Make the tailored QoS the new default. */
135     status = DDS_DomainParticipant_set_default_topic_qos(
136         participant, reliable_topic_qos);
137     checkStatus(status, "DDS_DomainParticipant_set_default_topic_qos");
138
139     /* Use the changed policy when defining the ChatMessage topic */
140     chatMessageTopic = DDS_DomainParticipant_create_topic(
```

```
141         participant,
142         "Chat_ChatMessage",
143         chatMessageTypeName,
144         reliable_topic_qos,
145         NULL,
146         DDS_ANY_STATUS);
147     checkHandle(
148         chatMessageTopic,
149         "DDS_DomainParticipant_create_topic (ChatMessage)");
150 
151     /* Set the DurabilityQosPolicy to TRANSIENT. */
152     setting_topic_qos = DDS_TopicQos__alloc();
153     checkHandle(setting_topic_qos, "DDS_TopicQos__alloc");
154     status = DDS_DomainParticipant_get_default_topic_qos(
155         participant, setting_topic_qos);
156     checkStatus(status, "DDS_DomainParticipant_get_default_topic_qos");
157     setting_topic_qos->durability.kind = DDS_TRANSIENT_DURABILITY_QOS;
158 
159     /* Create the NameService Topic. */
160     nameServiceTopic = DDS_DomainParticipant_create_topic(
161         participant,
162         "Chat_NameService",
163         nameServiceTypeName,
164         setting_topic_qos,
165         NULL,
166         DDS_ANY_STATUS);
167     checkHandle(nameServiceTopic, "DDS_DomainParticipant_create_topic");
168 
169     /* Adapt the default SubscriberQos to read from the
170        "ChatRoom" Partition. */
171     partitionName = "ChatRoom";
172     sub_qos = DDS_SubscriberQos__alloc();
173     checkHandle(sub_qos, "DDS_SubscriberQos__alloc");
174     status = DDS_DomainParticipant_get_default_subscriber_qos(
175         participant, sub_qos);
176     checkStatus(status, "DDS_DomainParticipant_get_default_subscriber_qos");
177     sub_qos->partition.name._length = 1;
178     sub_qos->partition.name._maximum = 1;
179     sub_qos->partition.name._buffer = DDS_StringSeq_allocbuf (1);
180     checkHandle(sub_qos->partition.name._buffer, "DDS_StringSeq_allocbuf");
181     sub_qos->partition.name._buffer[0] =
182         DDS_string_alloc(strlen(partitionName) + 1);
183     checkHandle(sub_qos->partition.name._buffer[0], "DDS_string_alloc");
184     strcpy (sub_qos->partition.name._buffer[0], partitionName);
185 
186     /* Create a Subscriber for the UserLoad application. */
187     chatSubscriber = DDS_DomainParticipant_create_subscriber(
188         participant, sub_qos, NULL, DDS_ANY_STATUS);
189     checkHandle(chatSubscriber, "DDS_DomainParticipant_create_subscriber");
190 
191     /* Create a DataReader for the NameService Topic
192        (using the appropriate QoS). */
193     nameServer = DDS_Subscriber_create_datareader(
194         chatSubscriber,
195         nameServiceTopic,
196         DDS_DATAREADER_QOS_USE_TOPIC_QOS,
197         NULL,
198         DDS_ANY_STATUS);
199     checkHandle(nameServer, "DDS_Subscriber_create_datareader (NameService)");
200 
201     /* Adapt the DataReaderQos for the ChatMessageDataReader
```

```
202        to keep track of all messages. */
203     message_qos = DDS_DataReaderQos__alloc();
204     checkHandle(message_qos, "DDS_DataReaderQos__alloc");
205     status = DDS_Subscriber_get_default_datareader_qos(
206         chatSubscriber, message_qos);
207     checkStatus(status, "DDS_Subscriber_get_default_datareader_qos");
208     status = DDS_Subscriber_copy_from_topic_qos(
209         chatSubscriber, message_qos, reliable_topic_qos);
210     checkStatus(status, "DDS_Subscriber_copy_from_topic_qos");
211     message_qos->history.kind = DDS_KEEP_ALL_HISTORY_QOS;
212
213     /* Create a DataReader for the ChatMessage Topic
214        (using the appropriate QoS). */
215     loadAdmin = DDS_Subscriber_create_datareader(
216         chatSubscriber,
217         chatMessageTopic,
218         message_qos,
219         NULL,
220         DDS_ANY_STATUS);
221     checkHandle(loadAdmin, "DDS_Subscriber_create_datareader (ChatMessage)");
222
223     /* Initialize the Query Arguments. */
224     args._length = 1;
225     args._maximum = 1;
226     args._buffer = DDS_StringSeq_allocbuf(1);
227     checkHandle(args._buffer, "DDS_StringSeq_allocbuf");
228     args._buffer[0] = DDS_string_alloc (12);  // Enough for max size numbers.
229     checkHandle(args._buffer[0], "DDS_string_alloc");
230     sprintf(args._buffer[0], "%d", 0);
231
232     /* Create a QueryCondition that will contain all messages
233        with userID = ownID */
234     singleUser = DDS_DataReader_create_querycondition(
235         loadAdmin,
236         DDS_ANY_SAMPLE_STATE,
237         DDS_ANY_VIEW_STATE,
238         DDS_ANY_INSTANCE_STATE,
239         "userID=%0",
240         &args);
241     checkHandle(
242         singleUser,
243         "DDS_DataReader_create_querycondition (singleUser Query)");
244
245     /* Create a ReadCondition that will contain new users only */
246     newUser = DDS_DataReader_create_readcondition(
247         nameServer,
248         DDS_NOT_READ_SAMPLE_STATE,
249         DDS_NEW_VIEW_STATE,
250         DDS_ALIVE_INSTANCE_STATE);
251     checkHandle(newUser, "DDS_DataReader_create_readcondition (newUser)");
252
253     /* Obtain a StatusCondition that triggers only when
254        a Writer changes Liveliness */
255     leftUser = DDS_DataReader_get_statuscondition(loadAdmin);
256     checkHandle(leftUser, "DDS_DataReader_get_statuscondition");
257     status = DDS_StatusCondition_set_enabled_statuses(
258         leftUser, DDS_LIVELINESS_CHANGED_STATUS);
259     checkStatus(status, "DDS_StatusCondition_set_enabled_statuses");
260
261     /* Create a bare guard which will be used to close the room */
262     escape = DDS_GuardCondition__alloc();
```

```
263     checkHandle(escape, "DDS_GuardCondition__alloc");
264
265     /* Create a waitset and add the ReadConditions */
266     userLoadWS = DDS_WaitSet__alloc();
267     checkHandle(userLoadWS, "DDS_WaitSet__alloc");
268     status = DDS_WaitSet_attach_condition(userLoadWS, newUser);
269     checkStatus(status, "DDS_WaitSet_attach_condition (newUser)");
270     status = DDS_WaitSet_attach_condition(userLoadWS, leftUser);
271     checkStatus(status, "DDS_WaitSet_attach_condition (leftUser)");
272     status = DDS_WaitSet_attach_condition(userLoadWS, escape);
273     checkStatus(status, "DDS_WaitSet_attach_condition (escape)");
274
275     /* Initialize and pre-allocate the GuardList used to obtain
276        the triggered Conditions. */
277     guardList = DDS_ConditionSeq__alloc();
278     checkHandle(guardList, "DDS_ConditionSeq__alloc");
279     guardList->_maximum = 3;
280     guardList->_length = 0;
281     guardList->_buffer = DDS_ConditionSeq_allocbuf(3);
282     checkHandle(guardList->_buffer, "DDS_ConditionSeq_allocbuf");
283
284     /* Remove all known Users that are not currently active. */
285     status = Chat_NameServiceDataReader_take(
286         nameServer,
287         &nsList,
288         &infoSeq,
289         DDS_LENGTH_UNLIMITED,
290         DDS_ANY_SAMPLE_STATE,
291         DDS_ANY_VIEW_STATE,
292         DDS_NOT_ALIVE_INSTANCE_STATE);
293     checkStatus(status, "Chat_NameServiceDataReader_take");
294     status = Chat_NameServiceDataReader_return_loan(
295         nameServer, &nsList, &infoSeq);
296     checkStatus(status, "Chat_NameServiceDataReader_return_loan");
297
298     /* Start the sleeper thread. */
299     pthread_create (&tid, NULL, delayedEscape, NULL);
300
301     while (!closed) {
302         /* Wait until at least one of the Conditions in the
303            waitset triggers. */
304         status = DDS_WaitSet_wait(userLoadWS, guardList, &timeout);
305         checkStatus(status, "DDS_WaitSet_wait");
306
307         /* Walk over all guards to display information */
308         for (i = 0; i < guardList->_length; i++) {
309             guard = guardList->_buffer[i];
310             if (guard == newUser) {
311                 /* The newUser ReadCondition contains data */
312                 status = Chat_NameServiceDataReader_read_w_condition(
313                     nameServer,
314                     &nsList,
315                     &infoSeq,
316                     DDS_LENGTH_UNLIMITED,
317                     newUser);
318                 checkStatus(
319                     status, "Chat_NameServiceDataReader_read_w_condition");
320
321                 for (j = 0; j < nsList._length; j++) {
322                     printf ("New user: %s\n", nsList._buffer[j].name);
323                 }
```

```
324                 status = Chat_NameServiceDataReader_return_loan(
325                     nameServer, &nsList, &infoSeq);
326                 checkStatus(status, "Chat_NameServiceDataReader_return_loan");
327
328             } else if (guard == leftUser) {
329                 /* Some liveliness has changed (either a DataWriter joined
330                    or a DataWriter left) */
331                 status = DDS_DataReader_get_liveliness_changed_status(
332                     loadAdmin, &livChangStatus);
333                 checkStatus(
334                     status, "DDS_DataReader_get_liveliness_changed_status");
335                 if (livChangStatus.alive_count < prevCount) {
336                     /* A user has left the ChatRoom, since a DataWriter lost
337                        its liveliness. Take the effected users so they will
338                        not appear in the list later on. */
339                     status = Chat_NameServiceDataReader_take(
340                         nameServer,
341                         &nsList,
342                         &infoSeq,
343                         DDS_LENGTH_UNLIMITED,
344                         DDS_ANY_SAMPLE_STATE,
345                         DDS_ANY_VIEW_STATE,
346                         DDS_NOT_ALIVE_NO_WRITERS_INSTANCE_STATE);
347                     checkStatus(status, "Chat_NameServiceDataReader_take");
348
349                     for (j = 0; j < nsList._length; j++) {
350                         /* re-apply query arguments */
351                         sprintf(
352                             args._buffer[0],
353                             "%d",
354                             nsList._buffer[j].userID);
355                         status = DDS_QueryCondition_set_query_parameters(
356                             singleUser, &args);
357                         checkStatus(
358                            status, "DDS_QueryCondition_set_query_parameters");
359
360                         /* Read this users history */
361                         status = Chat_ChatMessageDataReader_take_w_condition(
362                             loadAdmin,
363                             &msgList,
364                             &infoSeq2,
365                             DDS_LENGTH_UNLIMITED,
366                             singleUser);
367                         checkStatus(
368                             status,
369                             "Chat_ChatMessageDataReader_take_w_condition");
370
371                         /* Display the user and his history */
372                         printf (
373                             "Departed user %s has sent %d messages\n",
374                             nsList._buffer[j].name,
375                             msgList._length);
376                         status = Chat_ChatMessageDataReader_return_loan(
377                             loadAdmin, &msgList, &infoSeq2);
378                         checkStatus(
379                             status, "Chat_ChatMessageDataReader_return_loan");
380                     }
381                     status = Chat_NameServiceDataReader_return_loan(
382                         nameServer, &nsList, &infoSeq);
383                     checkStatus(
384                         status, "Chat_NameServiceDataReader_return_loan");
```

```
385                 }
386                 prevCount = livChangStatus.alive_count;
387
388             } else if (guard == escape) {
389                 printf ("UserLoad has terminated.\n");
390                 closed = 1;
391             }
392             else
393             {
394                 assert(0);
395             };
396         } /* for */
397     } /* while (!closed) */
398
399     /* Remove all Conditions from the WaitSet. */
400     status = DDS_WaitSet_detach_condition(userLoadWS, escape);
401     checkStatus(status, "DDS_WaitSet_detach_condition (escape)");
402     status = DDS_WaitSet_detach_condition(userLoadWS, leftUser);
403     checkStatus(status, "DDS_WaitSet_detach_condition (leftUser)");
404     status = DDS_WaitSet_detach_condition(userLoadWS, newUser);
405     checkStatus(status, "DDS_WaitSet_detach_condition (newUser)");
406
407     /* Free all resources */
408     DDS_free(guardList);
409     DDS_free(args._buffer);
410     DDS_free(userLoadWS);
411     DDS_free(escape);
412     DDS_free(setting_topic_qos);
413     DDS_free(reliable_topic_qos);
414     DDS_free(nameServiceTypeName);
415     DDS_free(chatMessageTypeName);
416     DDS_free(nameServiceTS);
417     DDS_free(chatMessageTS);
418     status = DDS_DomainParticipant_delete_contained_entities(participant);
419     checkStatus(status, "DDS_DomainParticipant_delete_contained_entities");
420     status = DDS_DomainParticipantFactory_delete_participant(
421         DDS_TheParticipantFactory,
422         participant);
423     checkStatus(status, "DDS_DomainParticipantFactory_delete_participant");
424
425     return 0;
426 }
```

Appendix

B C++ Language Examples' Code

This appendix lists the complete C++ source code for the examples provided in the C++ version of the OpenSplice DDS tutorial.

Chat.idl

```
/************************************************************************
 *
 * Copyright (c) 2006
 * PrismTech Ltd.
 * All rights Reserved.
 *
 * LOGICAL_NAME:    Chat.idl
 * FUNCTION:        OpenSplice DDS Tutorial example code.
 * MODULE:          Tutorial for the C++ programming language.
 * DATE             june 2006.
 ************************************************************************
 *
 * This file contains the data definitions for the tutorial examples.
 *
 ***/

module Chat {

    const long MAX_NAME = 32;
    typedef string<MAX_NAME> nameType;

    struct ChatMessage {
        long      userID;          // owner of message
        long      index;           // message number
        string    content;         // message body
    };
#pragma keylist ChatMessage userID

    struct NameService {
        long     userID;           // unique user identification
        nameType name;             // name of the user
    };
#pragma keylist NameService userID

    struct NamedMessage {
        long     userID;           // unique user identification
        nameType userName;         // user name
        long     index;            // message number
        string   content;          // message body
    };
#pragma keylist NamedMessage userID

};
```

CheckStatus.h

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    CheckStatus.h
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C++ programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the headers for the error handling operations.
14  *
15  ***/
16
17 #ifndef __CHECKSTATUS_H__
18 #define __CHECKSTATUS_H__
19
20 #include "ccpp_dds_dcps.h"
21 #include <iostream>
22
23 using namespace std;
24
25 /**
26  * Returns the name of an error code.
27  **/
28 char *getErrorName(DDS::ReturnCode_t status);
29
30 /**
31  * Check the return status for errors. If there is an error, then terminate.
32  **/
33 void checkStatus(DDS::ReturnCode_t status, const char *info);
34
35 /**
36  * Check whether a valid handle has been returned. If not, then terminate.
37  **/
38 void checkHandle(void *handle, char *info);
39
40 #endif
```

CheckStatus.cpp

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    CheckStatus.cpp
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C++ programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
```

```
 * This file contains the implementation for the error handling operations.
 *
 ***/

#include "CheckStatus.h"

/* Array to hold the names for all ReturnCodes. */
char *RetCodeName[13] = {
    "DDS_RETCODE_OK",
    "DDS_RETCODE_ERROR",
    "DDS_RETCODE_UNSUPPORTED",
    "DDS_RETCODE_BAD_PARAMETER",
    "DDS_RETCODE_PRECONDITION_NOT_MET",
    "DDS_RETCODE_OUT_OF_RESOURCES",
    "DDS_RETCODE_NOT_ENABLED",
    "DDS_RETCODE_IMMUTABLE_POLICY",
    "DDS_RETCODE_INCONSISTENT_POLICY",
    "DDS_RETCODE_ALREADY_DELETED",
    "DDS_RETCODE_TIMEOUT",
    "DDS_RETCODE_NO_DATA",
    "DDS_RETCODE_ILLEGAL_OPERATION" };

/**
 * Returns the name of an error code.
 **/
char *getErrorName(DDS::ReturnCode_t status)
{
    return RetCodeName[status];
}

/**
 * Check the return status for errors. If there is an error, then terminate.
 **/
void checkStatus(
    DDS::ReturnCode_t status,
    const char *info ) {


    if (status != DDS::RETCODE_OK && status != DDS::RETCODE_NO_DATA) {
        cerr << "Error in " << info << ": " << getErrorName(status) << endl;
        exit (0);
    }
}

/**
 * Check whether a valid handle has been returned. If not, then terminate.
 **/
void checkHandle(
    void *handle,
    char *info ) {

     if (!handle) {
        cerr << "Error in " << info <<
            ": Creation failed: invalid handle" << endl;
        exit (0);
     }
}
```

Chatter.cpp

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    Chatter.cpp
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C++ programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the implementation for the 'Chatter' executable.
14  *
15  ***/
16 #include <string>
17 #include <sstream>
18 #include <iostream>
19 #include <unistd.h>
20 #include "ccpp_dds_dcps.h"
21 #include "CheckStatus.h"
22 #include "ccpp_Chat.h"
23
24 #define MAX_MSG_LEN 256
25 #define NUM_MSG 10
26 #define TERMINATION_MESSAGE -1
27
28 using namespace DDS;
29 using namespace Chat;
30
31 int
32 main (
33     int argc,
34     char *argv[])
35 {
36     /* Generic DDS entities */
37     DomainParticipantFactory_var    dpf;
38     DomainParticipant_var           participant;
39     Topic_var                       chatMessageTopic;
40     Topic_var                       nameServiceTopic;
41     Publisher_var                   chatPublisher;
42     DataWriter_ptr                  parentWriter;
43
44     /* QosPolicy holders */
45     TopicQos                        reliable_topic_qos;
46     TopicQos                        setting_topic_qos;
47     PublisherQos                    pub_qos;
48     DataWriterQos                   dw_qos;
49
50     /* DDS Identifiers */
51     DomainId_t                      domain = NULL;
52     InstanceHandle_t                userHandle;
53     ReturnCode_t                    status;
54
55     /* Type-specific DDS entities */
56     ChatMessageTypeSupport_var      chatMessageTS;
57     NameServiceTypeSupport_var      nameServiceTS;
58     ChatMessageDataWriter_var       talker;
```

```
59      NameServiceDataWriter_var       nameServer;
60
61      /* Sample definitions */
62      ChatMessage                     *msg;   /* Example on Heap */
63      NameService                     ns;     /* Example on Stack */
64
65      /* Others */
66      int                             ownID = 1;
67      int                             i;
68      char                            *chatterName = NULL;
69      const char                      *partitionName = "ChatRoom";
70      char                            *chatMessageTypeName = NULL;
71      char                            *nameServiceTypeName = NULL;
72      ostringstream                   buf;
73
74
75
76      /* Options: Chatter [ownID [name]] */
77      if (argc > 1) {
78          istringstream args(argv[1]);
79          args >> ownID;
80          if (argc > 2) {
81              chatterName = argv[2];
82          }
83      }
84
85      /* Create a DomainParticipantFactory and a DomainParticipant
86         (using Default QoS settings. */
87      dpf = DomainParticipantFactory::get_instance ();
88      checkHandle(dpf.in(), "DDS::DomainParticipantFactory::get_instance");
89      participant = dpf->create_participant(
90          domain, PARTICIPANT_QOS_DEFAULT, NULL, ANY_STATUS);
91      checkHandle(
92          participant.in(),
93          "DDS::DomainParticipantFactory::create_participant");
94
95      /* Register the required datatype for ChatMessage. */
96      chatMessageTS = new ChatMessageTypeSupport();
97      checkHandle(chatMessageTS.in(), "new ChatMessageTypeSupport");
98      chatMessageTypeName = chatMessageTS->get_type_name();
99      status = chatMessageTS->register_type(
100         participant.in(),
101         chatMessageTypeName);
102     checkStatus(status, "Chat::ChatMessageTypeSupport::register_type");
103
104     /* Register the required datatype for NameService. */
105     nameServiceTS = new NameServiceTypeSupport();
106     checkHandle(nameServiceTS.in(), "new NameServiceTypeSupport");
107     nameServiceTypeName =  nameServiceTS->get_type_name();
108     status = nameServiceTS->register_type(
109         participant.in(),
110         nameServiceTypeName);
111     checkStatus(status, "Chat::NameServiceTypeSupport::register_type");
112
113     /* Set the ReliabilityQosPolicy to RELIABLE. */
114     status = participant->get_default_topic_qos(reliable_topic_qos);
115     checkStatus(status, "DDS::DomainParticipant::get_default_topic_qos");
116     reliable_topic_qos.reliability.kind = RELIABLE_RELIABILITY_QOS;
117
118     /* Make the tailored QoS the new default. */
119     status = participant->set_default_topic_qos(reliable_topic_qos);
```

```
120     checkStatus(status, "DDS::DomainParticipant::set_default_topic_qos");
121
122     /* Use the changed policy when defining the ChatMessage topic */
123     chatMessageTopic = participant->create_topic(
124         "Chat_ChatMessage",
125         chatMessageTypeName,
126         reliable_topic_qos,
127         NULL,
128         ANY_STATUS);
129     checkHandle(
130         chatMessageTopic.in(),
131         "DDS::DomainParticipant::create_topic (ChatMessage)");
132
133     /* Set the DurabilityQosPolicy to TRANSIENT. */
134     status = participant->get_default_topic_qos(setting_topic_qos);
135     checkStatus(status, "DDS::DomainParticipant::get_default_topic_qos");
136     setting_topic_qos.durability.kind = TRANSIENT_DURABILITY_QOS;
137
138     /* Create the NameService Topic. */
139     nameServiceTopic = participant->create_topic(
140         "Chat_NameService",
141         nameServiceTypeName,
142         setting_topic_qos,
143         NULL,
144         ANY_STATUS);
145     checkHandle(
146         nameServiceTopic.in(),
147         "DDS::DomainParticipant::create_topic (NameService)");
148
149     /* Adapt the default PublisherQos to write into the
150        "ChatRoom" Partition. */
151     status = participant->get_default_publisher_qos (pub_qos);
152     checkStatus(status, "DDS::DomainParticipant::get_default_publisher_qos");
153     pub_qos.partition.name.length(1);
154     pub_qos.partition.name[0] = partitionName;
155
156     /* Create a Publisher for the chatter application. */
157     chatPublisher = participant->create_publisher(pub_qos, NULL, ANY_STATUS);
158     checkHandle(
159         chatPublisher.in(), "DDS::DomainParticipant::create_publisher");
160
161     /* Create a DataWriter for the ChatMessage Topic
162        (using the appropriate QoS). */
163     parentWriter = chatPublisher->create_datawriter(
164         chatMessageTopic.in(),
165         DATAWRITER_QOS_USE_TOPIC_QOS,
166         NULL,
167         ANY_STATUS);
168     checkHandle(
169         parentWriter, "DDS::Publisher::create_datawriter (chatMessage)");
170
171     /* Narrow the abstract parent into its typed representative. */
172     talker = ChatMessageDataWriter::_narrow(parentWriter);
173     checkHandle(talker.in(), "Chat::ChatMessageDataWriter::_narrow");
174
175     /* Create a DataWriter for the NameService Topic
176        (using the appropriate QoS). */
177     status = chatPublisher->get_default_datawriter_qos(dw_qos);
178     checkStatus(status, "DDS::Publisher::get_default_datawriter_qos");
179     status = chatPublisher->copy_from_topic_qos(dw_qos, setting_topic_qos);
180     checkStatus(status, "DDS::Publisher::copy_from_topic_qos");
```

```
181     dw_qos.writer_data_lifecycle.autodispose_unregistered_instances = FALSE;
182     parentWriter = chatPublisher->create_datawriter(
183         nameServiceTopic.in(),
184         dw_qos,
185         NULL,
186         ANY_STATUS);
187     checkHandle(
188         parentWriter, "DDS::Publisher::create_datawriter (NameService)");
189
190     /* Narrow the abstract parent into its typed representative. */
191     nameServer = NameServiceDataWriter::_narrow(parentWriter);
192     checkHandle(nameServer.in(), "Chat::NameServiceDataWriter::_narrow");
193
194     /* Initialize the NameServer attributes located on stack. */
195     ns.userID = ownID;
196     if (chatterName) {
197         ns.name = CORBA::string_dup(chatterName);
198     } else {
199         buf << "Chatter " << ownID;
200         ns.name = CORBA::string_dup( buf.str().c_str() );
201     }
202
203     /* Write the user-information into the system
204        (registering the instance implicitly). */
205     status = nameServer->write(ns, HANDLE_NIL);
206     checkStatus(status, "Chat::ChatMessageDataWriter::write");
207
208     /* Initialize the chat messages on Heap. */
209     msg = new ChatMessage();
210     checkHandle(msg, "new ChatMessage");
211     msg->userID = ownID;
212     msg->index = 0;
213     buf.str( string("") );
214     if (ownID == TERMINATION_MESSAGE) {
215         buf << "Termination message.";
216     } else {
217         buf << "Hi there, I will send you " << NUM_MSG << " more messages.";
218     }
219     msg->content = CORBA::string_dup( buf.str().c_str() );
220     cout << "Writing message: \"" << msg->content  << "\"" << endl;
221
222     /* Register a chat message for this user
223        (pre-allocating resources for it!!) */
224     userHandle = talker->register_instance(*msg);
225
226     /* Write a message using the pre-generated instance handle. */
227     status = talker->write(*msg, userHandle);
228     checkStatus(status, "Chat::ChatMessageDataWriter::write");
229
230     sleep (1); /* do not run so fast! */
231
232     /* Write any number of messages, re-using the existing
233        string-buffer: no leak!!. */
234     for (i = 1; i <= NUM_MSG && ownID != TERMINATION_MESSAGE; i++) {
235         buf.str( string("") );
236         msg->index = i;
237         buf << "Message no. " << i;
238         msg->content = CORBA::string_dup( buf.str().c_str() );
239         cout << "Writing message: \"" << msg->content << "\"" << endl;
240         status = talker->write(*msg, userHandle);
241         checkStatus(status, "Chat::ChatMessageDataWriter::write");
```

```
242         sleep (1); /* do not run so fast! */
243     }
244
245     /* Leave the room by disposing and unregistering the message instance. */
246     status = talker->dispose(*msg, userHandle);
247     checkStatus(status, "Chat::ChatMessageDataWriter::dispose");
248     status = talker->unregister_instance(*msg, userHandle);
249     checkStatus(status, "Chat::ChatMessageDataWriter::unregister_instance");
250
251     /* Also unregister our name. */
252     status = nameServer->unregister_instance(ns, HANDLE_NIL);
253     checkStatus(status, "Chat::NameServiceDataWriter::unregister_instance");
254
255     /* Release the data-samples. */
256     delete msg; // msg allocated on heap: explicit de-allocation required!!
257
258     /* Remove the DataWriters */
259     status = chatPublisher->delete_datawriter( talker.in() );
260     checkStatus(status, "DDS::Publisher::delete_datawriter (talker)");
261
262     status = chatPublisher->delete_datawriter( nameServer.in() );
263     checkStatus(status, "DDS::Publisher::delete_datawriter (nameServer)");
264
265     /* Remove the Publisher. */
266     status = participant->delete_publisher( chatPublisher.in() );
267     checkStatus(status, "DDS::DomainParticipant::delete_publisher");
268
269     /* Remove the Topics. */
270     status = participant->delete_topic( nameServiceTopic.in() );
271     checkStatus(
272         status, "DDS::DomainParticipant::delete_topic (nameServiceTopic)");
273
274     status = participant->delete_topic( chatMessageTopic.in() );
275     checkStatus(
276         status, "DDS::DomainParticipant::delete_topic (chatMessageTopic)");
277
278     /* Remove the type-names. */
279     CORBA::string_free(chatMessageTypeName);
280     CORBA::string_free(nameServiceTypeName);
281
282     /* Remove the DomainParticipant. */
283     status = dpf->delete_participant( participant.in() );
284     checkStatus(status, "DDS::DomainParticipantFactory::delete_participant");
285
286     return 0;
287 }
```

MessageBoard.cpp

```
1  /*********************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    MessageBoard.cpp
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C++ programming language.
```

```
10  * DATE             june 2007.
11  ******************************************************************
12  *
13  * This file contains the implementation for the 'MessageBoard' executable.
14  *
15  ***/
16
17 #include <iostream>
18 #include <string.h>
19 #include <unistd.h>
20
21 #include "ccpp_dds_dcps.h"
22 #include "CheckStatus.h"
23 #include "ccpp_Chat.h"
24 #include "multitopic.h"
25
26 using namespace DDS;
27 using namespace Chat;
28
29
30
31 #define TERMINATION_MESSAGE -1
32
33
34
35 int
36 main (
37     int argc,
38     char *argv[])
39 {
40     /* Generic DDS entities */
41     DomainParticipantFactory_var    dpf;
42     DomainParticipant_ptr           parentDP;
43     ExtDomainParticipant_var        participant;
44     Topic_var                       chatMessageTopic;
45     Topic_var                       nameServiceTopic;
46     TopicDescription_var            namedMessageTopic;
47     Subscriber_var                  chatSubscriber;
48     DataReader_ptr                  parentReader;
49
50     /* Type-specific DDS entities */
51     ChatMessageTypeSupport_var      chatMessageTS;
52     NameServiceTypeSupport_var      nameServiceTS;
53     NamedMessageTypeSupport_var     namedMessageTS;
54     NamedMessageDataReader_var      chatAdmin;
55     NamedMessageSeq_var             msgSeq = new NamedMessageSeq();
56     SampleInfoSeq_var               infoSeq = new SampleInfoSeq();
57
58     /* QosPolicy holders */
59     TopicQos                        reliable_topic_qos;
60     TopicQos                        setting_topic_qos;
61     SubscriberQos                   sub_qos;
62     DDS::StringSeq                  parameterList;
63
64     /* DDS Identifiers */
65     DomainId_t                      domain = NULL;
66     ReturnCode_t                    status;
67
68     /* Others */
69     bool                            terminated = FALSE;
70     const char *                    partitionName = "ChatRoom";
```

```
71      char  *                         chatMessageTypeName = NULL;
72      char  *                         nameServiceTypeName = NULL;
73      char  *                         namedMessageTypeName = NULL;
74
75      /* Options: MessageBoard [ownID] */
76      /* Messages having owner ownID will be ignored */
77      parameterList.length(1);
78
79      if (argc > 1) {
80          parameterList[0] = CORBA::string_dup(argv[1]);
81      }
82      else
83      {
84          parameterList[0] = "0";
85      }
86
87      /* Create a DomainParticipantFactory and a DomainParticipant
88         (using Default QoS settings. */
89      dpf = DomainParticipantFactory::get_instance();
90      checkHandle(dpf.in(), "DDS::DomainParticipantFactory::get_instance");
91      parentDP = dpf->create_participant (
92          domain,
93          PARTICIPANT_QOS_DEFAULT,
94          NULL,
95          ANY_STATUS);
96      checkHandle(
97          parentDP, "DDS::DomainParticipantFactory::create_participant");
98
99      /* Narrow the normal participant to its extended representative */
100     participant = ExtDomainParticipantImpl::_narrow(parentDP);
101     checkHandle(participant.in(), "DDS::ExtDomainParticipant::_narrow");
102
103     /* Register the required datatype for ChatMessage. */
104     chatMessageTS = new ChatMessageTypeSupport();
105     checkHandle(chatMessageTS.in(), "new ChatMessageTypeSupport");
106     chatMessageTypeName = chatMessageTS->get_type_name();
107     status = chatMessageTS->register_type(
108         participant.in(),
109         chatMessageTypeName);
110     checkStatus(status, "Chat::ChatMessageTypeSupport::register_type");
111
112     /* Register the required datatype for NameService. */
113     nameServiceTS = new NameServiceTypeSupport();
114     checkHandle(nameServiceTS.in(), "new NameServiceTypeSupport");
115     nameServiceTypeName =  nameServiceTS->get_type_name();
116     status = nameServiceTS->register_type(
117         participant.in(),
118         nameServiceTypeName);
119     checkStatus(status, "Chat::NameServiceTypeSupport::register_type");
120
121     /* Register the required datatype for NamedMessage. */
122     namedMessageTS = new NamedMessageTypeSupport();
123     checkHandle(namedMessageTS.in(), "new NamedMessageTypeSupport");
124     namedMessageTypeName = namedMessageTS->get_type_name();
125     status = namedMessageTS->register_type(
126         participant.in(),
127         namedMessageTypeName);
128     checkStatus(status, "Chat::NamedMessageTypeSupport::register_type");
129
130     /* Set the ReliabilityQosPolicy to RELIABLE. */
131     status = participant->get_default_topic_qos(reliable_topic_qos);
```

```
132     checkStatus(status, "DDS::DomainParticipant::get_default_topic_qos");
133     reliable_topic_qos.reliability.kind = DDS::RELIABLE_RELIABILITY_QOS;
134
135     /* Make the tailored QoS the new default. */
136     status = participant->set_default_topic_qos(reliable_topic_qos);
137     checkStatus(status, "DDS::DomainParticipant::set_default_topic_qos");
138
139     /* Use the changed policy when defining the ChatMessage topic */
140     chatMessageTopic = participant->create_topic(
141         "Chat_ChatMessage",
142         chatMessageTypeName,
143         reliable_topic_qos,
144         NULL,
145         ANY_STATUS);
146     checkHandle(
147         chatMessageTopic.in(),
148         "DDS::DomainParticipant::create_topic (ChatMessage)");
149
150     /* Set the DurabilityQosPolicy to TRANSIENT. */
151     status = participant->get_default_topic_qos(setting_topic_qos);
152     checkStatus(status, "DDS::DomainParticipant::get_default_topic_qos");
153     setting_topic_qos.durability.kind = DDS::TRANSIENT_DURABILITY_QOS;
154
155     /* Create the NameService Topic. */
156     nameServiceTopic = participant->create_topic(
157         "Chat_NameService",
158         nameServiceTypeName,
159         setting_topic_qos,
160         NULL,
161         ANY_STATUS);
162     checkHandle(
163         nameServiceTopic.in(), "DDS::DomainParticipant::create_topic");
164
165     /* Create a multitopic that substitutes the userID with its
166        corresponding userName. */
167     namedMessageTopic = participant->create_simulated_multitopic(
168         "Chat_NamedMessage",
169         namedMessageTypeName,
170         "SELECT userID, name AS userName, index, content "
171             "FROM Chat_NameService NATURAL JOIN Chat_ChatMessage "
172             "WHERE userID <> %0",
173         parameterList);
174     checkHandle(
175         namedMessageTopic.in(),
176         "DDS::ExtDomainParticipant::create_simulated_multitopic");
177
178     /* Adapt the default SubscriberQos to read from the
179        "ChatRoom" Partition. */
180     status = participant->get_default_subscriber_qos (sub_qos);
181     checkStatus(
182         status, "DDS::DomainParticipant::get_default_subscriber_qos");
183     sub_qos.partition.name.length(1);
184     sub_qos.partition.name[0] = partitionName;
185
186     /* Create a Subscriber for the MessageBoard application. */
187     chatSubscriber = participant->create_subscriber(
188         sub_qos, NULL, ANY_STATUS);
189     checkHandle(
190         chatSubscriber.in(), "DDS::DomainParticipant::create_subscriber");
191
192     /* Create a DataReader for the NamedMessage Topic
```

```
193        (using the appropriate QoS). */
194     parentReader = chatSubscriber->create_datareader(
195         namedMessageTopic.in(),
196         DATAREADER_QOS_USE_TOPIC_QOS,
197         NULL,
198         ANY_STATUS);
199     checkHandle(parentReader, "DDS::Subscriber::create_datareader");
200
201     /* Narrow the abstract parent into its typed representative. */
202     chatAdmin = Chat::NamedMessageDataReader::_narrow(parentReader);
203     checkHandle(chatAdmin.in(), "Chat::NamedMessageDataReader::_narrow");
204
205     /* Print a message that the MessageBoard has opened. */
206     cout << "MessageBoard has opened: send a ChatMessage with "
207         "userID = -1 to close it...." << endl << endl;
208
209     while (!terminated) {
210         /* Note: using read does not remove the samples from
211            unregistered instances from the DataReader. This means
212            that the DataRase would use more and more resources.
213            That's why we use take here instead. */
214
215         status = chatAdmin->take(
216             msgSeq,
217             infoSeq,
218             LENGTH_UNLIMITED,
219             ANY_SAMPLE_STATE,
220             ANY_VIEW_STATE,
221             ALIVE_INSTANCE_STATE );
222         checkStatus(status, "Chat::NamedMessageDataReader::take");
223
224         for (CORBA::ULong i = 0; i < msgSeq->length(); i++) {
225             NamedMessage *msg = &(msgSeq[i]);
226             if (msg->userID == TERMINATION_MESSAGE) {
227                 cout << "Termination message received: exiting..." << endl;
228                 terminated = TRUE;
229             } else {
230                 cout << msg->userName << ": " << msg->content << endl;
231             }
232         }
233
234         status = chatAdmin->return_loan(msgSeq, infoSeq);
235         checkStatus(status, "Chat::ChatMessageDataReader::return_loan");
236
237         /* Sleep for some amount of time, as not to consume
238            too much CPU cycles. */
239         usleep(100000);
240     }
241
242     /* Remove the DataReader */
243     status = chatSubscriber->delete_datareader(chatAdmin.in());
244     checkStatus(status, "DDS::Subscriber::delete_datareader");
245
246     /* Remove the Subscriber. */
247     status = participant->delete_subscriber(chatSubscriber.in());
248     checkStatus(status, "DDS::DomainParticipant::delete_subscriber");
249
250     /* Remove the Topics. */
251     status = participant->delete_simulated_multitopic(
252         namedMessageTopic.in());
253     checkStatus(
```

```
254         status, "DDS::ExtDomainParticipant::delete_simulated_multitopic");
255
256     status = participant->delete_topic(nameServiceTopic.in());
257     checkStatus(
258         status, "DDS::DomainParticipant::delete_topic (nameServiceTopic)");
259
260     status = participant->delete_topic(chatMessageTopic.in());
261     checkStatus(
262         status, "DDS::DomainParticipant::delete_topic (chatMessageTopic)");
263
264     /* De-allocate the type-names. */
265     CORBA::string_free(namedMessageTypeName);
266     CORBA::string_free(nameServiceTypeName);
267     CORBA::string_free(chatMessageTypeName);
268
269     /* Remove the DomainParticipant. */
270     status = dpf->delete_participant(participant.in());
271     checkStatus(status, "DDS::DomainParticipantFactory::delete_participant");
272
273     exit(0);
274 }
```

multitopic.h

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    multitopic.h
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C++ programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the headers for all operations required to simulate
14  * the MultiTopic behavior.
15  *
16  ***/
17
18 #include <string>
19
20 #include "ccpp_dds_dcps.h"
21 #include "ccpp_Chat.h"
22 #include "orb_abstraction.h"
23
24
25 namespace DDS {
26
27 class DataReaderListenerImpl : public virtual DDS::DataReaderListener {
28
29     /* Caching variables */
30     CORBA::Long                         previous;
31     std::string                         userName;
32
33 public:
34     /* Type-specific DDS entities */
```

```
35      Chat::ChatMessageDataReader_var     chatMessageDR;
36      Chat::NameServiceDataReader_var     nameServiceDR;
37      Chat::NamedMessageDataWriter_var    namedMessageDW;
38
39      /* Query related stuff */
40      DDS::QueryCondition_var             nameFinder;
41      DDS::StringSeq                      nameFinderParams;
42
43
44      /* Constructor */
45      DataReaderListenerImpl();
46
47      /* Callback method implementation. */
48      virtual void on_requested_deadline_missed (
49          DDS::DataReader_ptr reader,
50          const DDS::RequestedDeadlineMissedStatus & status
51      ) THROW_ORB_EXCEPTIONS;
52
53      virtual void on_requested_incompatible_qos (
54          DDS::DataReader_ptr reader,
55          const DDS::RequestedIncompatibleQosStatus & status
56      ) THROW_ORB_EXCEPTIONS;
57
58      virtual void on_sample_rejected (
59          DDS::DataReader_ptr reader,
60          const DDS::SampleRejectedStatus & status
61      ) THROW_ORB_EXCEPTIONS;
62
63      virtual void on_liveliness_changed (
64          DDS::DataReader_ptr reader,
65          const DDS::LivelinessChangedStatus & status
66      ) THROW_ORB_EXCEPTIONS;
67
68      virtual void on_data_available (
69          DDS::DataReader_ptr reader
70      ) THROW_ORB_EXCEPTIONS;
71
72      virtual void on_subscription_matched (
73          DDS::DataReader_ptr reader,
74          const DDS::SubscriptionMatchedStatus & status
75      ) THROW_ORB_EXCEPTIONS;
76
77      virtual void on_sample_lost (
78          DDS::DataReader_ptr reader,
79          const DDS::SampleLostStatus & status
80      ) THROW_ORB_EXCEPTIONS;
81  };
82
83  class ExtDomainParticipantImpl;
84
85  typedef ExtDomainParticipantImpl *ExtDomainParticipant_ptr;
86
87  class ExtDomainParticipant_var {
88      ExtDomainParticipant_ptr ptr_;
89  public:
90      ExtDomainParticipant_var() : ptr_(NULL){};
91      ~ExtDomainParticipant_var();
92      ExtDomainParticipant_var & operator=(
93          const DDS::ExtDomainParticipant_ptr ep);
94      DDS::ExtDomainParticipant_ptr operator->() const;
95      operator const DDS::DomainParticipant_ptr() const;
```

```
96      DDS::DomainParticipant_ptr in() const;
97  };
98
99
100 class ExtDomainParticipantImpl
101     : public virtual DDS::DomainParticipant,
102       public LOCAL_REFCOUNTED_OBJECT
103 {
104     /***
105      * Attributes
106      ***/
107
108     // Encapsulated DomainParticipant.
109     DDS::DomainParticipant_var          realParticipant;
110
111     /*Implementation for DataReaderListener */
112     DDS::DataReaderListenerImpl         *msgListener;
113
114     /* Generic DDS entities */
115     DDS::Topic_var                      chatMessageTopic;
116     DDS::Topic_var                      nameServiceTopic;
117     DDS::ContentFilteredTopic_var       filteredMessageTopic;
118     DDS::Topic_var                      namedMessageTopic;
119     DDS::Subscriber_var                 multiSub;
120     DDS::Publisher_var                  multiPub;
121
122     /***
123      * Operations
124      ***/
125 public:
126
127     // Simulating a narrow operation.
128     static ExtDomainParticipant_ptr _narrow (
129         DDS::DomainParticipant_ptr obj
130     );
131
132     // Simulating an in() parameter where a DomainParticipant is expected.
133     DDS::DomainParticipant_ptr in();
134
135     // Constructor
136     ExtDomainParticipantImpl(DomainParticipant_ptr participant);
137
138     virtual DDS::Topic_ptr create_simulated_multitopic (
139         const char * name,
140         const char * type_name,
141         const char * subscription_expression,
142         const DDS::StringSeq & expression_parameters
143     );
144
145     virtual DDS::ReturnCode_t delete_simulated_multitopic (
146         DDS::TopicDescription_ptr a_topic
147     );
148
149     virtual DDS::ReturnCode_t enable (
150     ) THROW_ORB_EXCEPTIONS;
151
152     virtual DDS::StatusCondition_ptr get_statuscondition (
153     ) THROW_ORB_EXCEPTIONS;
154
155     virtual DDS::StatusKindMask get_status_changes (
156     ) THROW_ORB_EXCEPTIONS;
```

```
157
158     virtual DDS::InstanceHandle_t get_instance_handle (
159     ) THROW_ORB_EXCEPTIONS;
160
161     virtual DDS::Publisher_ptr create_publisher (
162         const DDS::PublisherQos & qos,
163         DDS::PublisherListener_ptr a_listener,
164         DDS::StatusMask mask
165     ) THROW_ORB_EXCEPTIONS;
166
167     virtual DDS::ReturnCode_t delete_publisher (
168         DDS::Publisher_ptr p
169     ) THROW_ORB_EXCEPTIONS;
170
171     virtual DDS::Subscriber_ptr create_subscriber (
172         const DDS::SubscriberQos & qos,
173         DDS::SubscriberListener_ptr a_listener,
174         DDS::StatusMask mask
175     ) THROW_ORB_EXCEPTIONS;
176
177     virtual DDS::ReturnCode_t delete_subscriber (
178         DDS::Subscriber_ptr s
179     ) THROW_ORB_EXCEPTIONS;
180
181     virtual DDS::Subscriber_ptr get_builtin_subscriber (
182     ) THROW_ORB_EXCEPTIONS;
183
184     virtual DDS::Topic_ptr create_topic (
185         const char * topic_name,
186         const char * type_name,
187         const DDS::TopicQos & qos,
188         DDS::TopicListener_ptr a_listener,
189         DDS::StatusMask mask
190     ) THROW_ORB_EXCEPTIONS;
191
192     virtual DDS::ReturnCode_t delete_topic (
193         DDS::Topic_ptr a_topic
194     ) THROW_ORB_EXCEPTIONS;
195
196     virtual DDS::Topic_ptr find_topic (
197         const char * topic_name,
198         const DDS::Duration_t & timeout
199     ) THROW_ORB_EXCEPTIONS;
200
201     virtual DDS::TopicDescription_ptr lookup_topicdescription (
202         const char * name
203     ) THROW_ORB_EXCEPTIONS;
204
205     virtual DDS::ContentFilteredTopic_ptr create_contentfilteredtopic (
206         const char * name,
207         DDS::Topic_ptr related_topic,
208         const char * filter_expression,
209         const DDS::StringSeq & filter_parameters
210     ) THROW_ORB_EXCEPTIONS;
211
212     virtual DDS::ReturnCode_t delete_contentfilteredtopic (
213         DDS::ContentFilteredTopic_ptr a_contentfilteredtopic
214     ) THROW_ORB_EXCEPTIONS;
215
216     virtual DDS::MultiTopic_ptr create_multitopic (
217         const char * name,
```

```
218         const char * type_name,
219         const char * subscription_expression,
220         const DDS::StringSeq & expression_parameters
221     ) THROW_ORB_EXCEPTIONS;
222
223     virtual DDS::ReturnCode_t delete_multitopic (
224         DDS::MultiTopic_ptr a_multitopic
225     ) THROW_ORB_EXCEPTIONS;
226
227     virtual DDS::ReturnCode_t delete_contained_entities (
228     ) THROW_ORB_EXCEPTIONS;
229
230     virtual DDS::ReturnCode_t set_qos (
231         const DDS::DomainParticipantQos & qos
232     ) THROW_ORB_EXCEPTIONS;
233
234     virtual DDS::ReturnCode_t get_qos (
235         DDS::DomainParticipantQos & qos
236     ) THROW_ORB_EXCEPTIONS;
237
238     virtual DDS::ReturnCode_t set_listener (
239         DDS::DomainParticipantListener_ptr a_listener,
240         DDS::StatusKindMask mask
241     ) THROW_ORB_EXCEPTIONS;
242
243     virtual DDS::DomainParticipantListener_ptr get_listener (
244     ) THROW_ORB_EXCEPTIONS;
245
246     virtual DDS::ReturnCode_t ignore_participant (
247         DDS::InstanceHandle_t handle
248     ) THROW_ORB_EXCEPTIONS;
249
250     virtual DDS::ReturnCode_t ignore_topic (
251         DDS::InstanceHandle_t handle
252     ) THROW_ORB_EXCEPTIONS;
253
254     virtual DDS::ReturnCode_t ignore_publication (
255         DDS::InstanceHandle_t handle
256     ) THROW_ORB_EXCEPTIONS;
257
258     virtual DDS::ReturnCode_t ignore_subscription (
259         DDS::InstanceHandle_t handle
260     ) THROW_ORB_EXCEPTIONS;
261
262     virtual char * get_domain_id (
263     ) THROW_ORB_EXCEPTIONS;
264
265     virtual DDS::ReturnCode_t assert_liveliness (
266     ) THROW_ORB_EXCEPTIONS;
267
268     virtual DDS::ReturnCode_t set_default_publisher_qos (
269         const DDS::PublisherQos & qos
270     ) THROW_ORB_EXCEPTIONS;
271
272     virtual DDS::ReturnCode_t get_default_publisher_qos (
273         DDS::PublisherQos & qos
274     ) THROW_ORB_EXCEPTIONS;
275
276     virtual DDS::ReturnCode_t set_default_subscriber_qos (
277         const DDS::SubscriberQos & qos
278     ) THROW_ORB_EXCEPTIONS;
```

```
279
280     virtual DDS::ReturnCode_t get_default_subscriber_qos (
281         DDS::SubscriberQos & qos
282     ) THROW_ORB_EXCEPTIONS;
283
284     virtual DDS::ReturnCode_t set_default_topic_qos (
285         const DDS::TopicQos & qos
286     ) THROW_ORB_EXCEPTIONS;
287
288     virtual DDS::ReturnCode_t get_default_topic_qos (
289         DDS::TopicQos & qos
290     ) THROW_ORB_EXCEPTIONS;
291
292     virtual DDS::ReturnCode_t get_discovered_participants (
293         DDS::InstanceHandleSeq & participant_handles
294     ) THROW_ORB_EXCEPTIONS;
295
296     virtual DDS::ReturnCode_t get_discovered_participant_data (
297         DDS::InstanceHandle_t participant_handle,
298         DDS::ParticipantBuiltinTopicData & participant_data
299     ) THROW_ORB_EXCEPTIONS;
300
301     virtual DDS::ReturnCode_t get_discovered_topics (
302         DDS::InstanceHandleSeq & topic_handles
303     ) THROW_ORB_EXCEPTIONS;
304
305     virtual DDS::ReturnCode_t get_discovered_topic_data (
306         DDS::InstanceHandle_t topic_handle,
307         DDS::TopicBuiltinTopicData & topic_data
308     ) THROW_ORB_EXCEPTIONS;
309
310     virtual CORBA::Boolean contains_entity (
311         DDS::InstanceHandle_t a_handle
312     ) THROW_ORB_EXCEPTIONS;
313
314     virtual DDS::ReturnCode_t get_current_time (
315         DDS::Time_t & current_time
316     ) THROW_ORB_EXCEPTIONS;
317 };
318
319 };
```

multitopic.cpp

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    multitopic.cpp
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C++ programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the headers for all operations required to simulate
14  * the MultiTopic behavior.
```

```
 *
 ***/

#include "multitopic.h"
#include "CheckStatus.h"
#include <sstream>

DDS::DataReaderListenerImpl::DataReaderListenerImpl() : previous(0x80000000) {
    nameFinderParams.length(1);
}

void
DDS::DataReaderListenerImpl::on_requested_deadline_missed (
    DDS::DataReader_ptr reader,
    const DDS::RequestedDeadlineMissedStatus & status
) THROW_ORB_EXCEPTIONS { };

void
DDS::DataReaderListenerImpl::on_requested_incompatible_qos (
    DDS::DataReader_ptr reader,
    const DDS::RequestedIncompatibleQosStatus & status
) THROW_ORB_EXCEPTIONS { };

void
DDS::DataReaderListenerImpl::on_sample_rejected (
    DDS::DataReader_ptr reader,
    const DDS::SampleRejectedStatus & status
) THROW_ORB_EXCEPTIONS { };

void
DDS::DataReaderListenerImpl::on_liveliness_changed (
    DDS::DataReader_ptr reader,
    const DDS::LivelinessChangedStatus & status
) THROW_ORB_EXCEPTIONS { };

void
DDS::DataReaderListenerImpl::on_subscription_matched (
    DDS::DataReader_ptr reader,
    const DDS::SubscriptionMatchedStatus & status
) THROW_ORB_EXCEPTIONS { };

void
DDS::DataReaderListenerImpl::on_sample_lost (
    DDS::DataReader_ptr reader,
    const DDS::SampleLostStatus & status
) THROW_ORB_EXCEPTIONS { };

void
DDS::DataReaderListenerImpl::on_data_available (
    DDS::DataReader_ptr reader
) THROW_ORB_EXCEPTIONS {
    Chat::ChatMessageSeq               msgSeq;
    Chat::NameServiceSeq               nameSeq;
    DDS::SampleInfoSeq                 infoSeq1;
    DDS::SampleInfoSeq                 infoSeq2;
    DDS::ReturnCode_t                  status;

    /* Take all messages. */
    status = chatMessageDR->take(
        msgSeq,
        infoSeq1,
```

```
76          DDS::LENGTH_UNLIMITED,
77          DDS::ANY_SAMPLE_STATE,
78          DDS::ANY_VIEW_STATE,
79          DDS::ANY_INSTANCE_STATE);
80      checkStatus(status, "Chat::ChatMessageDataReader::take");
81
82      /* For each message, extract the key-field and find
83         the corresponding name. */
84      for (CORBA::ULong i = 0; i < msgSeq.length(); i++)
85      {
86          if (infoSeq1[i].valid_data)
87          {
88              Chat::NamedMessage joinedSample;
89
90              /* Find the corresponding named message. */
91              if (msgSeq[i].userID != previous)
92              {
93                  ostringstream numberStr;
94                  previous = msgSeq[i].userID;
95                  numberStr << previous;
96                  nameFinderParams[0UL] = numberStr.str().c_str();
97                  status = nameFinder->set_query_parameters(nameFinderParams);
98               checkStatus(status, "DDS::QueryCondition::set_query_parameters");
99                  status = nameServiceDR->read_w_condition(
100                     nameSeq,
101                     infoSeq2,
102                     DDS::LENGTH_UNLIMITED,
103                     nameFinder.in());
104                 checkStatus(
105                     status, "Chat::NameServiceDataReader::read_w_condition");
106
107                 /* Extract Name (there should only be one result). */
108                 if (status == DDS::RETCODE_NO_DATA)
109                 {
110                     ostringstream msg;
111                     msg << "Name not found!! id = " << previous;
112                     userName = msg.str();
113                 }
114                 else
115                 {
116                     userName = nameSeq[0].name;
117                 }
118
119                 /* Release the name sample again. */
120                 status = nameServiceDR->return_loan(nameSeq, infoSeq2);
121               checkStatus(status, "Chat::NameServiceDataReader::return_loan");
122         }
123             /* Write merged Topic with userName instead of userID. */
124             joinedSample.userName = userName.c_str();
125             joinedSample.userID = msgSeq[i].userID;
126             joinedSample.index = msgSeq[i].index;
127             joinedSample.content = msgSeq[i].content;
128             status = namedMessageDW->write(joinedSample, DDS::HANDLE_NIL);
129             checkStatus(status, "Chat::NamedMessageDataWriter::write");
130         }
131     }
132     status = chatMessageDR->return_loan(msgSeq, infoSeq1);
133     checkStatus(status, "Chat::ChatMessageDataReader::return_loan");
134 };
135
136
```

```
DDS::ExtDomainParticipant_ptr
DDS::ExtDomainParticipantImpl::_narrow(DDS::DomainParticipant_ptr obj) {
    return new DDS::ExtDomainParticipantImpl(obj);
};

DDS::DomainParticipant_ptr
DDS::ExtDomainParticipantImpl::in() {
    return realParticipant.in();
};


DDS::ExtDomainParticipantImpl::ExtDomainParticipantImpl(
    DDS::DomainParticipant_ptr participant
) {
    realParticipant = DDS::DomainParticipant::_duplicate(participant);
};



DDS::Topic_ptr
DDS::ExtDomainParticipantImpl::create_simulated_multitopic (
    const char * name,
    const char * type_name,
    const char * subscription_expression,
    const DDS::StringSeq & expression_parameters)
{
    /* Type-specific DDS entities */
    Chat::ChatMessageDataReader_ptr     chatMessageDR;
    Chat::NameServiceDataReader_ptr     nameServiceDR;
    Chat::NamedMessageDataWriter_ptr    namedMessageDW;

    /* Query related stuff */
    DDS::QueryCondition_ptr             nameFinder;

    /* QosPolicy holders */
    DDS::TopicQos                       namedMessageQos;
    DDS::SubscriberQos                  sub_qos;
    DDS::PublisherQos                   pub_qos;

    /* Others */
    DDS::DataReader_ptr                 parentReader;
    DDS::DataWriter_ptr                 parentWriter;
    char                                *nameFinderExpr;
    const char                          *partitionName = "ChatRoom";
    DDS::ReturnCode_t                   status;

    /* Lookup both components that constitute the multi-topic. */
    chatMessageTopic = realParticipant->find_topic(
        "Chat_ChatMessage", DDS::DURATION_INFINITY);
    checkHandle(
        chatMessageTopic.in(),
        "DDS::DomainParticipant::find_topic (Chat_ChatMessage)");

    nameServiceTopic = realParticipant->find_topic(
        "Chat_NameService", DDS::DURATION_INFINITY);
    checkHandle(
        nameServiceTopic.in(),
        "DDS::DomainParticipant::find_topic (Chat_NameService)");

    /* Create a ContentFilteredTopic to filter out our own ChatMessages. */
    filteredMessageTopic = realParticipant->create_contentfilteredtopic(
```

```
198         "Chat_FilteredMessage",
199         chatMessageTopic.in(),
200         "userID <> %0",
201         expression_parameters);
202     checkHandle(
203         filteredMessageTopic.in(),
204         "DDS::DomainParticipant::create_contentfilteredtopic");
205
206
207     /* Adapt the default SubscriberQos to read from the
208        "ChatRoom" Partition. */
209     status = realParticipant->get_default_subscriber_qos (sub_qos);
210     checkStatus(status, "DDS::DomainParticipant::get_default_subscriber_qos");
211     sub_qos.partition.name.length(1);
212     sub_qos.partition.name[0] = partitionName;
213
214     /* Create a private Subscriber for the multitopic simulator. */
215     multiSub = realParticipant->create_subscriber(
216         sub_qos, NULL, DDS::ANY_STATUS);
217     checkHandle(
218         multiSub.in(),
219         "DDS::DomainParticipant::create_subscriber (for multitopic)");
220
221     /* Create a DataReader for the FilteredMessage Topic
222        (using the appropriate QoS). */
223     parentReader = multiSub->create_datareader(
224         filteredMessageTopic.in(),
225         DATAREADER_QOS_USE_TOPIC_QOS,
226         NULL,
227         DDS::ANY_STATUS);
228     checkHandle(
229         parentReader,
230         "DDS::Subscriber::create_datareader (ChatMessage)");
231
232     /* Narrow the abstract parent into its typed representative. */
233     chatMessageDR = Chat::ChatMessageDataReader::_narrow(parentReader);
234     checkHandle(chatMessageDR, "Chat::ChatMessageDataReader::_narrow");
235
236     /* Allocate the DataReaderListener Implementation. */
237     msgListener = new DDS::DataReaderListenerImpl();
238     checkHandle(msgListener, "new DDS::DataReaderListenerImpl");
239
240     /* Attach the DataReaderListener to the DataReader, only enabling
241        the data_available event. */
242     status = chatMessageDR->set_listener(
243         msgListener, DDS::DATA_AVAILABLE_STATUS);
244     checkStatus(status, "DDS::DataReader_set_listener");
245
246     /* Create a DataReader for the nameService Topic
247        (using the appropriate QoS). */
248     parentReader = multiSub->create_datareader(
249         nameServiceTopic.in(),
250         DATAREADER_QOS_USE_TOPIC_QOS,
251         NULL,
252         DDS::ANY_STATUS);
253     checkHandle(
254         parentReader, "DDS::Subscriber::create_datareader (NameService)");
255
256     /* Narrow the abstract parent into its typed representative. */
257     nameServiceDR = Chat::NameServiceDataReader::_narrow(parentReader);
258     checkHandle(nameServiceDR, "Chat::NameServiceDataReader::_narrow");
```

```
259
260     /* Define the SQL expression (using a parameterized value). */
261     nameFinderExpr = "userID = %0";
262
263     /* Create a QueryCondition to only read corresponding nameService
264        information by key-value. */
265     nameFinder = nameServiceDR->create_querycondition(
266         DDS::ANY_SAMPLE_STATE,
267         DDS::ANY_VIEW_STATE,
268         DDS::ANY_INSTANCE_STATE,
269         nameFinderExpr,
270         expression_parameters);
271     checkHandle(
272         nameFinder, "DDS::DataReader::create_querycondition (nameFinder)");
273
274     /* Create the Topic that simulates the multi-topic
275        (use Qos from chatMessage).*/
276     status = chatMessageTopic->get_qos(namedMessageQos);
277     checkStatus(status, "DDS::Topic::get_qos");
278
279     /* Create the NamedMessage Topic whose samples simulate the MultiTopic */
280     namedMessageTopic = realParticipant->create_topic(
281         "Chat_NamedMessage",
282         type_name,
283         namedMessageQos,
284         NULL,
285         DDS::ANY_STATUS);
286     checkHandle(
287     namedMessageTopic.in(),
288     "DDS::DomainParticipant::create_topic (NamedMessage)");
289
290     /* Adapt the default PublisherQos to write into the
291        "ChatRoom" Partition. */
292     status = realParticipant->get_default_publisher_qos(pub_qos);
293     checkStatus(status, "DDS::DomainParticipant::get_default_publisher_qos");
294     pub_qos.partition.name.length(1);
295     pub_qos.partition.name[0] = partitionName;
296
297     /* Create a private Publisher for the multitopic simulator. */
298     multiPub = realParticipant->create_publisher(
299         pub_qos, NULL, DDS::ANY_STATUS);
300     checkHandle(
301         multiPub.in(),
302         "DDS::DomainParticipant::create_publisher (for multitopic)");
303
304     /* Create a DataWriter for the multitopic. */
305     parentWriter = multiPub->create_datawriter(
306         namedMessageTopic.in(),
307         DATAWRITER_QOS_USE_TOPIC_QOS,
308         NULL,
309         DDS::ANY_STATUS);
310     checkHandle(
311         parentWriter, "DDS::Publisher::create_datawriter (NamedMessage)");
312
313     /* Narrow the abstract parent into its typed representative. */
314     namedMessageDW = Chat::NamedMessageDataWriter::_narrow(parentWriter);
315     checkHandle(namedMessageDW, "Chat::NamedMessageDataWriter::_narrow");
316
317     /* Store the relevant Entities in our Listener. */
318     msgListener->chatMessageDR = chatMessageDR;
319     msgListener->nameServiceDR = nameServiceDR;
```

```
320     msgListener->namedMessageDW = namedMessageDW;
321     msgListener->nameFinder = nameFinder;
322
323     /* Return the simulated Multitopic. */
324     return DDS::Topic::_duplicate( namedMessageTopic.in() );
325 };
326
327 DDS::ReturnCode_t
328 DDS::ExtDomainParticipantImpl::delete_simulated_multitopic(
329     DDS::TopicDescription_ptr smt
330 )
331 {
332     DDS::ReturnCode_t status;
333
334     /* Remove the DataWriter */
335     status = multiPub->delete_datawriter(msgListener->namedMessageDW.in());
336     checkStatus(status, "DDS::Publisher::delete_datawriter");
337
338     /* Remove the Publisher. */
339     status = realParticipant->delete_publisher(multiPub.in());
340     checkStatus(status, "DDS::DomainParticipant::delete_publisher");
341
342     /* Remove the QueryCondition. */
343     status = msgListener->nameServiceDR->delete_readcondition(
344         msgListener->nameFinder.in());
345     checkStatus(status, "DDS::DataReader::delete_readcondition");
346
347     /* Remove the DataReaders. */
348     status = multiSub->delete_datareader(msgListener->nameServiceDR.in());
349     checkStatus(status, "DDS::Subscriber::delete_datareader");
350     status = multiSub->delete_datareader(msgListener->chatMessageDR.in());
351     checkStatus(status, "DDS::Subscriber::delete_datareader");
352
353     /* Remove the DataReaderListener. */
354     CORBA::release(msgListener);
355
356     /* Remove the Subscriber. */
357     status = realParticipant->delete_subscriber(multiSub.in());
358     checkStatus(status, "DDS::DomainParticipant::delete_subscriber");
359
360     /* Remove the ContentFilteredTopic. */
361     status = realParticipant->delete_contentfilteredtopic(
362         filteredMessageTopic.in());
363     checkStatus(
364         status, "DDS::DomainParticipant::delete_contentfilteredtopic");
365
366     /* Remove all other topics. */
367     status = realParticipant->delete_topic(namedMessageTopic.in());
368     checkStatus(
369         status, "DDS::DomainParticipant::delete_topic (namedMessageTopic)");
370     status = realParticipant->delete_topic(nameServiceTopic.in());
371     checkStatus(
372         status, "DDS::DomainParticipant::delete_topic (nameServiceTopic)");
373     status = realParticipant->delete_topic(chatMessageTopic.in());
374     checkStatus(
375         status, "DDS::DomainParticipant::delete_topic (chatMessageTopic)");
376
377     return status;
378 };
379
380
```

```
381
382 DDS::ReturnCode_t
383 DDS::ExtDomainParticipantImpl::enable (
384 ) THROW_ORB_EXCEPTIONS {
385     return realParticipant->enable();
386 };
387
388 DDS::StatusCondition_ptr
389 DDS::ExtDomainParticipantImpl::get_statuscondition (
390 ) THROW_ORB_EXCEPTIONS {
391     return realParticipant->get_statuscondition();
392 };
393
394 DDS::StatusKindMask
395 DDS::ExtDomainParticipantImpl::get_status_changes (
396 ) THROW_ORB_EXCEPTIONS {
397     return realParticipant->get_status_changes();
398 };
399
400 DDS::InstanceHandle_t
401 DDS::ExtDomainParticipantImpl::get_instance_handle (
402 ) THROW_ORB_EXCEPTIONS {
403     return realParticipant->get_instance_handle();
404 };
405
406 DDS::Publisher_ptr
407 DDS::ExtDomainParticipantImpl::create_publisher (
408     const DDS::PublisherQos & qos,
409     DDS::PublisherListener_ptr a_listener,
410     DDS::StatusMask mask
411 ) THROW_ORB_EXCEPTIONS {
412     return realParticipant->create_publisher(qos, a_listener, mask);
413 };
414
415 DDS::ReturnCode_t
416 DDS::ExtDomainParticipantImpl::delete_publisher (
417     DDS::Publisher_ptr p
418 ) THROW_ORB_EXCEPTIONS {
419     return realParticipant->delete_publisher(p);
420 };
421
422 DDS::Subscriber_ptr
423 DDS::ExtDomainParticipantImpl::create_subscriber (
424     const DDS::SubscriberQos & qos,
425     DDS::SubscriberListener_ptr a_listener,
426     DDS::StatusMask mask
427 ) THROW_ORB_EXCEPTIONS {
428     return realParticipant->create_subscriber(qos, a_listener, mask);
429 };
430
431 DDS::ReturnCode_t
432 DDS::ExtDomainParticipantImpl::delete_subscriber (
433     DDS::Subscriber_ptr s
434 ) THROW_ORB_EXCEPTIONS {
435     return realParticipant->delete_subscriber(s);
436 };
437
438 DDS::Subscriber_ptr
439 DDS::ExtDomainParticipantImpl::get_builtin_subscriber (
440 ) THROW_ORB_EXCEPTIONS {
441     return realParticipant->get_builtin_subscriber();
```

```
};

DDS::Topic_ptr
DDS::ExtDomainParticipantImpl::create_topic (
    const char * topic_name,
    const char * type_name,
    const DDS::TopicQos & qos,
    DDS::TopicListener_ptr a_listener,
    DDS::StatusMask mask
) THROW_ORB_EXCEPTIONS {
    return realParticipant->create_topic(topic_name, type_name, qos,
a_listener, mask);
};

DDS::ReturnCode_t
DDS::ExtDomainParticipantImpl::delete_topic (
    DDS::Topic_ptr a_topic
) THROW_ORB_EXCEPTIONS {
    return realParticipant->delete_topic(a_topic);
};

DDS::Topic_ptr
DDS::ExtDomainParticipantImpl::find_topic (
    const char * topic_name,
    const DDS::Duration_t & timeout
) THROW_ORB_EXCEPTIONS {
    return realParticipant->find_topic(topic_name, timeout);
};

DDS::TopicDescription_ptr
DDS::ExtDomainParticipantImpl::lookup_topicdescription (
    const char * name
) THROW_ORB_EXCEPTIONS {
    return realParticipant->lookup_topicdescription(name);
};

DDS::ContentFilteredTopic_ptr
DDS::ExtDomainParticipantImpl::create_contentfilteredtopic (
    const char * name,
    DDS::Topic_ptr related_topic,
    const char * filter_expression,
    const DDS::StringSeq & filter_parameters
) THROW_ORB_EXCEPTIONS {
    return realParticipant->create_contentfilteredtopic(
        name,
        related_topic,
        filter_expression,
        filter_parameters);
};

DDS::ReturnCode_t
DDS::ExtDomainParticipantImpl::delete_contentfilteredtopic (
    DDS::ContentFilteredTopic_ptr a_contentfilteredtopic
) THROW_ORB_EXCEPTIONS {
    return realParticipant->delete_contentfilteredtopic(
        a_contentfilteredtopic);
};

DDS::MultiTopic_ptr
DDS::ExtDomainParticipantImpl::create_multitopic (
```

```
501     const char * name,
502     const char * type_name,
503     const char * subscription_expression,
504     const DDS::StringSeq & expression_parameters
505 ) THROW_ORB_EXCEPTIONS {
506     return realParticipant->create_multitopic(
507         name,
508         type_name,
509         subscription_expression,
510         expression_parameters);
511 };
512
513 DDS::ReturnCode_t
514 DDS::ExtDomainParticipantImpl::delete_multitopic (
515     DDS::MultiTopic_ptr a_multitopic
516 ) THROW_ORB_EXCEPTIONS {
517     return realParticipant->delete_multitopic(a_multitopic);
518 };
519
520 DDS::ReturnCode_t
521 DDS::ExtDomainParticipantImpl::delete_contained_entities (
522 ) THROW_ORB_EXCEPTIONS {
523     return realParticipant->delete_contained_entities();
524 };
525
526 DDS::ReturnCode_t
527 DDS::ExtDomainParticipantImpl::set_qos (
528     const DDS::DomainParticipantQos & qos
529 ) THROW_ORB_EXCEPTIONS {
530     return realParticipant->set_qos(qos);
531 };
532
533 DDS::ReturnCode_t
534 DDS::ExtDomainParticipantImpl::get_qos (
535     DDS::DomainParticipantQos & qos
536 ) THROW_ORB_EXCEPTIONS {
537     return realParticipant->get_qos(qos);
538 };
539
540 DDS::ReturnCode_t
541 DDS::ExtDomainParticipantImpl::set_listener (
542     DDS::DomainParticipantListener_ptr a_listener,
543     DDS::StatusKindMask mask
544 ) THROW_ORB_EXCEPTIONS {
545     return realParticipant->set_listener(a_listener, mask);
546 };
547
548 DDS::DomainParticipantListener_ptr
549 DDS::ExtDomainParticipantImpl::get_listener (
550 ) THROW_ORB_EXCEPTIONS {
551     return realParticipant->get_listener();
552 };
553
554 DDS::ReturnCode_t
555 DDS::ExtDomainParticipantImpl::ignore_participant (
556     DDS::InstanceHandle_t handle
557 ) THROW_ORB_EXCEPTIONS {
558     return realParticipant->ignore_participant(handle);
559 };
560
561 DDS::ReturnCode_t
```

```
562 DDS::ExtDomainParticipantImpl::ignore_topic (
563     DDS::InstanceHandle_t handle
564 ) THROW_ORB_EXCEPTIONS {
565     return realParticipant->ignore_topic(handle);
566 };
567
568 DDS::ReturnCode_t
569 DDS::ExtDomainParticipantImpl::ignore_publication (
570     DDS::InstanceHandle_t handle
571 ) THROW_ORB_EXCEPTIONS {
572     return realParticipant->ignore_publication(handle);
573 };
574
575 DDS::ReturnCode_t
576 DDS::ExtDomainParticipantImpl::ignore_subscription (
577     DDS::InstanceHandle_t handle
578 ) THROW_ORB_EXCEPTIONS {
579     return realParticipant->ignore_subscription(handle);
580 };
581
582 char *
583 DDS::ExtDomainParticipantImpl::get_domain_id (
584 ) THROW_ORB_EXCEPTIONS {
585     return realParticipant->get_domain_id();
586 };
587
588 DDS::ReturnCode_t
589 DDS::ExtDomainParticipantImpl::assert_liveliness (
590 ) THROW_ORB_EXCEPTIONS {
591     return realParticipant->assert_liveliness();
592 };
593
594 DDS::ReturnCode_t
595 DDS::ExtDomainParticipantImpl::set_default_publisher_qos (
596     const DDS::PublisherQos & qos
597 ) THROW_ORB_EXCEPTIONS {
598     return realParticipant->set_default_publisher_qos(qos);
599 };
600
601 DDS::ReturnCode_t
602 DDS::ExtDomainParticipantImpl::get_default_publisher_qos (
603     DDS::PublisherQos & qos
604 ) THROW_ORB_EXCEPTIONS {
605     return realParticipant->get_default_publisher_qos(qos);
606 };
607
608 DDS::ReturnCode_t
609 DDS::ExtDomainParticipantImpl::set_default_subscriber_qos (
610     const DDS::SubscriberQos & qos
611 ) THROW_ORB_EXCEPTIONS {
612     return realParticipant->set_default_subscriber_qos(qos);
613 };
614
615 DDS::ReturnCode_t
616 DDS::ExtDomainParticipantImpl::get_default_subscriber_qos (
617     DDS::SubscriberQos & qos
618 ) THROW_ORB_EXCEPTIONS {
619     return realParticipant->get_default_subscriber_qos(qos);
620 };
621
622 DDS::ReturnCode_t
```

```
623 DDS::ExtDomainParticipantImpl::set_default_topic_qos (
624     const DDS::TopicQos & qos
625 ) THROW_ORB_EXCEPTIONS {
626     return realParticipant->set_default_topic_qos(qos);
627 };
628
629 DDS::ReturnCode_t
630 DDS::ExtDomainParticipantImpl::get_default_topic_qos (
631     DDS::TopicQos & qos
632 ) THROW_ORB_EXCEPTIONS {
633     return realParticipant->get_default_topic_qos(qos);
634 };
635
636 DDS::ReturnCode_t
637 DDS::ExtDomainParticipantImpl::get_discovered_participants (
638     DDS::InstanceHandleSeq & participant_handles
639 ) THROW_ORB_EXCEPTIONS {
640     return realParticipant->get_discovered_participants(participant_handles);
641 };
642
643 DDS::ReturnCode_t
644 DDS::ExtDomainParticipantImpl::get_discovered_participant_data (
645     DDS::InstanceHandle_t participant_handle,
646     DDS::ParticipantBuiltinTopicData & participant_data
647 ) THROW_ORB_EXCEPTIONS {
648     return realParticipant->get_discovered_participant_data(
649         participant_handle, participant_data);
650 };
651
652 DDS::ReturnCode_t
653 DDS::ExtDomainParticipantImpl::get_discovered_topics (
654     DDS::InstanceHandleSeq & topic_handles
655 ) THROW_ORB_EXCEPTIONS {
656     return realParticipant->get_discovered_topics(topic_handles);
657 };
658
659 DDS::ReturnCode_t
660 DDS::ExtDomainParticipantImpl::get_discovered_topic_data (
661     DDS::InstanceHandle_t topic_handle,
662     DDS::TopicBuiltinTopicData & topic_data
663 ) THROW_ORB_EXCEPTIONS {
664     return realParticipant->get_discovered_topic_data(
665         topic_handle, topic_data);
666 };
667
668 CORBA::Boolean
669 DDS::ExtDomainParticipantImpl::contains_entity (
670     DDS::InstanceHandle_t a_handle
671 ) THROW_ORB_EXCEPTIONS {
672     return realParticipant->contains_entity(a_handle);
673 };
674
675 DDS::ReturnCode_t
676 DDS::ExtDomainParticipantImpl::get_current_time (
677     DDS::Time_t & current_time
678 ) THROW_ORB_EXCEPTIONS {
679     return realParticipant->get_current_time(current_time);
680 };
681
682 DDS::ExtDomainParticipant_var::~ExtDomainParticipant_var() {
683     CORBA::release(ptr_);
```

```
684 };
685
686 DDS::ExtDomainParticipant_var &
687 DDS::ExtDomainParticipant_var::operator=(
688     const DDS::ExtDomainParticipant_ptr ep
689 ) {
690     ptr_ = ep;
691     return *this;
692 };
693
694 DDS::ExtDomainParticipant_ptr
695 DDS::ExtDomainParticipant_var::operator->() const {
696     return ptr_;
697 };
698
699 DDS::ExtDomainParticipant_var::operator const
700 DDS::DomainParticipant_ptr() const {
701     return ptr_->in();
702 };
703
704 DDS::DomainParticipant_ptr DDS::ExtDomainParticipant_var::in() const {
705     return ptr_->in();
706 };
```

UserLoad.cpp

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    UserLoad.cpp
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the C++ programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the implementation for the 'UserLoad' executable.
14  *
15  ***/
16
17 #include <iostream>
18 #include <sstream>
19 #include <unistd.h>
20 #include <string.h>
21 #include <pthread.h>
22 #include <assert.h>
23
24 #include "ccpp_dds_dcps.h"
25 #include "CheckStatus.h"
26 #include "ccpp_Chat.h"
27
28 using namespace DDS;
29 using namespace Chat;
30
31 /* entities required by all threads. */
32 static DDS::GuardCondition_var          escape;
```

```
33
34 /* Sleeper thread: sleeps 60 seconds and then triggers the WaitSet. */
35 void *
36 delayedEscape(
37     void *arg)
38 {
39     DDS::ReturnCode_t status;
40
41     sleep(60);     /* wait for 60 sec. */
42     status = escape->set_trigger_value(TRUE);
43     checkStatus(status, "DDS::GuardCondition::set_trigger_value");
44
45     return NULL;
46 }
47
48 int
49 main (
50     int argc,
51     char *argv[])
52 {
53     /* Generic DDS entities */
54     DomainParticipant_var           participant;
55     Topic_var                       chatMessageTopic;
56     Topic_var                       nameServiceTopic;
57     Subscriber_var                  chatSubscriber;
58     DataReader_ptr                  parentReader;
59     QueryCondition_var              singleUser;
60     ReadCondition_var               newUser;
61     StatusCondition_var             leftUser;
62     WaitSet_var                     userLoadWS;
63     LivelinessChangedStatus         livChangStatus;
64
65     /* QosPolicy holders */
66     TopicQos                        setting_topic_qos;
67     TopicQos                        reliable_topic_qos;
68     SubscriberQos                   sub_qos;
69     DataReaderQos                   message_qos;
70
71     /* DDS Identifiers */
72     DomainId_t                      domain = NULL;
73     ReturnCode_t                    status;
74     ConditionSeq                    guardList;
75
76     /* Type-specific DDS entities */
77     ChatMessageTypeSupport_var      chatMessageTS;
78     NameServiceTypeSupport_var      nameServiceTS;
79     NameServiceDataReader_var       nameServer;
80     ChatMessageDataReader_var       loadAdmin;
81     ChatMessageSeq                  msgList;
82     NameServiceSeq                  nsList;
83     SampleInfoSeq                   infoSeq;
84     SampleInfoSeq                   infoSeq2;
85
86     /* Others */
87     StringSeq                       args;
88     char *                          chatMessageTypeName = NULL;
89     char *                          nameServiceTypeName = NULL;
90
91     bool                            closed = false;
92     CORBA::Long                     prevCount = 0;
93     pthread_t                       tid;
```

```
94
95      /* Create a DomainParticipant (using the 'TheParticipantFactory'
96         convenience macro). */
97      participant = TheParticipantFactory->create_participant (
98          domain,
99          PARTICIPANT_QOS_DEFAULT,
100         NULL,
101         ANY_STATUS);
102     checkHandle(
103     participant.in(), "DDS::DomainParticipantFactory::create_participant");
104
105     /* Register the required datatype for ChatMessage. */
106     chatMessageTS = new ChatMessageTypeSupport();
107     checkHandle(chatMessageTS.in(), "new ChatMessageTypeSupport");
108     chatMessageTypeName = chatMessageTS->get_type_name();
109     status = chatMessageTS->register_type(
110         participant.in(), chatMessageTypeName);
111     checkStatus(status, "Chat::ChatMessageTypeSupport::register_type");
112
113     /* Register the required datatype for NameService. */
114     nameServiceTS = new NameServiceTypeSupport();
115     checkHandle(nameServiceTS.in(), "new NameServiceTypeSupport");
116     nameServiceTypeName =  nameServiceTS->get_type_name();
117     status = nameServiceTS->register_type(
118         participant.in(), nameServiceTypeName);
119     checkStatus(status, "Chat::NameServiceTypeSupport::register_type");
120
121     /* Set the ReliabilityQosPolicy to RELIABLE. */
122     status = participant->get_default_topic_qos(reliable_topic_qos);
123     checkStatus(status, "DDS::DomainParticipant::get_default_topic_qos");
124     reliable_topic_qos.reliability.kind = RELIABLE_RELIABILITY_QOS;
125
126     /* Make the tailored QoS the new default. */
127     status = participant->set_default_topic_qos(reliable_topic_qos);
128     checkStatus(status, "DDS::DomainParticipant::set_default_topic_qos");
129
130     /* Use the changed policy when defining the ChatMessage topic */
131     chatMessageTopic = participant->create_topic(
132         "Chat_ChatMessage",
133         chatMessageTypeName,
134         reliable_topic_qos,
135         NULL,
136         ANY_STATUS);
137     checkHandle(
138         chatMessageTopic.in(),
139         "DDS::DomainParticipant::create_topic (ChatMessage)");
140
141     /* Set the DurabilityQosPolicy to TRANSIENT. */
142     status = participant->get_default_topic_qos(setting_topic_qos);
143     checkStatus(status, "DDS::DomainParticipant::get_default_topic_qos");
144     setting_topic_qos.durability.kind = TRANSIENT_DURABILITY_QOS;
145
146     /* Create the NameService Topic. */
147     nameServiceTopic = participant->create_topic(
148         "Chat_NameService",
149         nameServiceTypeName,
150         setting_topic_qos,
151         NULL,
152         ANY_STATUS);
153     checkHandle(
154         nameServiceTopic.in(), "DDS::DomainParticipant::create_topic");
```

```
155
156    /* Adapt the default SubscriberQos to read from the "ChatRoom" Partition. */
157     status = participant->get_default_subscriber_qos (sub_qos);
158     checkStatus(
159         status, "DDS::DomainParticipant::get_default_subscriber_qos");
160     sub_qos.partition.name.length(1);
161     sub_qos.partition.name[0UL] = "ChatRoom";
162
163     /* Create a Subscriber for the UserLoad application. */
164     chatSubscriber = participant->create_subscriber(
165         sub_qos, NULL, ANY_STATUS);
166     checkHandle(
167         chatSubscriber.in(), "DDS::DomainParticipant::create_subscriber");
168
169     /* Create a DataReader for the NameService Topic
170        (using the appropriate QoS). */
171     parentReader = chatSubscriber->create_datareader(
172         nameServiceTopic.in(),
173         DATAREADER_QOS_USE_TOPIC_QOS,
174         NULL,
175         ANY_STATUS);
176     checkHandle(
177         parentReader, "DDS::Subscriber::create_datareader (NameService)");
178
179     /* Narrow the abstract parent into its typed representative. */
180     nameServer = NameServiceDataReader::_narrow(parentReader);
181     checkHandle(nameServer.in(), "Chat::NameServiceDataReader::_narrow");
182
183     /* Adapt the DataReaderQos for the ChatMessageDataReader to
184        keep track of all messages. */
185     status = chatSubscriber->get_default_datareader_qos(message_qos);
186     checkStatus(status, "DDS::Subscriber::get_default_datareader_qos");
187     status = chatSubscriber->copy_from_topic_qos(
188         message_qos, reliable_topic_qos);
189     checkStatus(status, "DDS::Subscriber::copy_from_topic_qos");
190     message_qos.history.kind = KEEP_ALL_HISTORY_QOS;
191
192     /* Create a DataReader for the ChatMessage Topic (using the appropriate
QoS). */
193     parentReader = chatSubscriber->create_datareader(
194         chatMessageTopic.in(),
195         message_qos,
196         NULL,
197         ANY_STATUS);
198     checkHandle(
199         parentReader, "DDS::Subscriber::create_datareader (ChatMessage)");
200
201     /* Narrow the abstract parent into its typed representative. */
202     loadAdmin = ChatMessageDataReader::_narrow(parentReader);
203     checkHandle(loadAdmin.in(), "Chat::ChatMessageDataReader::_narrow");
204
205     /* Initialize the Query Arguments. */
206     args.length(1);
207     args[0UL] = "0";
208
209     /* Create a QueryCondition that will contain all messages
210        with userID=ownID */
211     singleUser = loadAdmin->create_querycondition(
212         ANY_SAMPLE_STATE,
213         ANY_VIEW_STATE,
```

```
214         ANY_INSTANCE_STATE,
215         "userID=%0",
216         args);
217     checkHandle(singleUser.in(), "DDS::DataReader::create_querycondition");
218
219     /* Create a ReadCondition that will contain new users only */
220     newUser = nameServer->create_readcondition(
221         NOT_READ_SAMPLE_STATE,
222         NEW_VIEW_STATE,
223         ALIVE_INSTANCE_STATE);
224     checkHandle(newUser.in(), "DDS::DataReader::create_readcondition");
225
226     /* Obtain a StatusCondition that triggers only when a
227        Writer changes Liveliness */
228     leftUser = loadAdmin->get_statuscondition();
229     checkHandle(leftUser.in(), "DDS::DataReader::get_statuscondition");
230     status = leftUser->set_enabled_statuses(LIVELINESS_CHANGED_STATUS);
231     checkStatus(status, "DDS::StatusCondition::set_enabled_statuses");
232
233     /* Create a bare guard which will be used to close the room */
234     escape = new GuardCondition();
235
236     /* Create a waitset and add the ReadConditions */
237     userLoadWS = new WaitSet();
238     status = userLoadWS->attach_condition(newUser.in());
239     checkStatus(status, "DDS::WaitSet::attach_condition (newUser)");
240     status = userLoadWS->attach_condition(leftUser.in());
241     checkStatus(status, "DDS::WaitSet::attach_condition (leftUser)");
242     status = userLoadWS->attach_condition(escape.in());
243     checkStatus(status, "DDS::WaitSet::attach_condition (escape)");
244
245     /* Initialize and pre-allocate the GuardList used to
246        obtain the triggered Conditions. */
247     guardList.length(3);
248
249
250     /* Remove all known Users that are not currently active. */
251     status = nameServer->take(
252         nsList,
253         infoSeq,
254         LENGTH_UNLIMITED,
255         ANY_SAMPLE_STATE,
256         ANY_VIEW_STATE,
257         NOT_ALIVE_INSTANCE_STATE);
258     checkStatus(status, "Chat::NameServiceDataReader::take");
259     status = nameServer->return_loan(nsList, infoSeq);
260     checkStatus(status, "Chat::NameServiceDataReader::return_loan");
261
262     /* Start the sleeper thread. */
263     pthread_create (&tid, NULL, delayedEscape, NULL);
264
265     while (!closed) {
266         /* Wait until at least one of the Conditions in the
267            waitset triggers. */
268         status = userLoadWS->wait(guardList, DURATION_INFINITY);
269         checkStatus(status, "DDS::WaitSet::wait");
270
271         /* Walk over all guards to display information */
272         for (CORBA::ULong i = 0; i < guardList.length(); i++) {
273             if ( guardList[i] == newUser.in() ) {
274                 /* The newUser ReadCondition contains data */
```

```
275                 status = nameServer->read_w_condition(
276                     nsList,
277                     infoSeq,
278                     LENGTH_UNLIMITED,
279                     newUser.in() );
280                 checkStatus(
281                     status, "Chat::NameServiceDataReader::read_w_condition");
282
283                 for (CORBA::ULong j = 0; j < nsList.length(); j++) {
284                     cout << "New user: " << nsList[j].name << endl;
285                 }
286                 status = nameServer->return_loan(nsList, infoSeq);
287                 checkStatus(
288                     status, "Chat::NameServiceDataReader::return_loan");
289
290             } else if ( guardList[i] == leftUser.in() ) {
291                 /* Some liveliness has changed (either a DataWriter joined
292                    or a DataWriter left) */
293                 status = loadAdmin->get_liveliness_changed_status(
294                     livChangStatus);
295                 checkStatus(
296                     status,
297                     "DDS::DataReader::get_liveliness_changed_status");
298                 if (livChangStatus.alive_count < prevCount) {
299                     /* A user has left the ChatRoom, since a DataWriter lost
300                        its liveliness. Take the effected users so they will
301                        not appear in the list later on. */
302                     status = nameServer->take(
303                         nsList,
304                         infoSeq,
305                         LENGTH_UNLIMITED,
306                         ANY_SAMPLE_STATE,
307                         ANY_VIEW_STATE,
308                         NOT_ALIVE_NO_WRITERS_INSTANCE_STATE);
309                     checkStatus(status, "Chat::NameServiceDataReader::take");
310
311                     for (CORBA::ULong j = 0; j < nsList.length(); j++) {
312                         /* re-apply query arguments */
313                         ostringstream numberString;
314                         numberString << nsList[j].userID;
315                         args[0UL] = numberString.str().c_str();
316                         status = singleUser->set_query_parameters(args);
317                         checkStatus(
318                             status,
319                             "DDS::QueryCondition::set_query_parameters");
320
321                         /* Read this users history */
322                         status = loadAdmin->take_w_condition(
323                             msgList,
324                             infoSeq2,
325                             LENGTH_UNLIMITED,
326                             singleUser.in() );
327                         checkStatus(
328                             status,
329                             "Chat::ChatMessageDataReader::take_w_condition");
330
331                         /* Display the user and his history */
332                         cout << "Departed user " << nsList[j].name <<
333                             " has sent " << msgList.length() <<
334                             " messages." << endl;
335                         status = loadAdmin->return_loan(msgList, infoSeq2);
```

```
336                         checkStatus(
337                             status,
338                             "Chat::ChatMessageDataReader::return_loan");
339                     }
340                     status = nameServer->return_loan(nsList, infoSeq);
341                     checkStatus(
342                         status, "Chat::NameServiceDataReader::return_loan");
343                 }
344                 prevCount = livChangStatus.alive_count;
345
346             } else if ( guardList[i] == escape.in() ) {
347                 cout << "UserLoad has terminated." << endl;
348                 closed = true;
349             }
350             else
351             {
352                 assert(0);
353             };
354         } /* for */
355     } /* while (!closed) */
356
357     /* Remove all Conditions from the WaitSet. */
358     status = userLoadWS->detach_condition( escape.in() );
359     checkStatus(status, "DDS::WaitSet::detach_condition (escape)");
360     status = userLoadWS->detach_condition( leftUser.in() );
361     checkStatus(status, "DDS::WaitSet::detach_condition (leftUser)");
362     status = userLoadWS->detach_condition( newUser.in() );
363     checkStatus(status, "DDS::WaitSet::detach_condition (newUser)");
364
365     /* Remove the type-names. */
366     CORBA::string_free(chatMessageTypeName);
367     CORBA::string_free(nameServiceTypeName);
368
369     /* Free all resources */
370     status = participant->delete_contained_entities();
371     checkStatus(status, "DDS::DomainParticipant::delete_contained_entities");
372     status = TheParticipantFactory->delete_participant( participant.in() );
373     checkStatus(status, "DDS::DomainParticipantFactory::delete_participant");
374
375     return 0;
376 }
```

Appendix

C Java Language Examples' Code

This appendix lists the complete Java source code for the examples provided in the Java version of the OpenSplice DDS tutorial.

Chat.idl

```
1  /************************************************************************
2   *
3   * Copyright (c) 2006
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    Chat.idl
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the Java programming language.
10  * DATE             june 2006.
11  ************************************************************************
12  *
13  * This file contains the data definitions for the tutorial examples.
14  *
15  ***/
16
17 module Chat {
18
19     const long MAX_NAME = 32;
20     typedef string<MAX_NAME> nameType;
21
22     struct ChatMessage {
23         long      userID;          // owner of message
24         long      index;           // message number
25         string    content;         // message body
26     };
27 #pragma keylist ChatMessage userID
28
29     struct NameService {
30         long     userID;           // unique user identification
31         nameType name;             // name of the user
32     };
33 #pragma keylist NameService userID
34
35     struct NamedMessage {
36         long     userID;           // unique user identification
37         nameType userName;         // user name
38         long     index;            // message number
39         string   content;          // message body
40     };
41 #pragma keylist NamedMessage userID
42
43 };
```

ErrorHandler.java

```
1  /************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    ErrorHandler.java
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the Java programming language.
10  * DATE             june 2007.
11  ************************************************************************
12  *
13  * This file contains the implementation for the error handling operations.
14  *
15  ***/
16
17 package chatroom;
18
19 import DDS.*;
20
21 public class ErrorHandler {
22
23    public static final int NR_ERROR_CODES = 13;
24
25    /* Array to hold the names for all ReturnCodes. */
26    public static String[] RetCodeName = new String[NR_ERROR_CODES];
27
28    static {
29       RetCodeName[0] = new String("DDS_RETCODE_OK");
30       RetCodeName[1] = new String("DDS_RETCODE_ERROR");
31       RetCodeName[2] = new String("DDS_RETCODE_UNSUPPORTED");
32       RetCodeName[3] = new String("DDS_RETCODE_BAD_PARAMETER");
33       RetCodeName[4] = new String("DDS_RETCODE_PRECONDITION_NOT_MET");
34       RetCodeName[5] = new String("DDS_RETCODE_OUT_OF_RESOURCES");
35       RetCodeName[6] = new String("DDS_RETCODE_NOT_ENABLED");
36       RetCodeName[7] = new String("DDS_RETCODE_IMMUTABLE_POLICY");
37       RetCodeName[8] = new String("DDS_RETCODE_INCONSISTENT_POLICY");
38       RetCodeName[9] = new String("DDS_RETCODE_ALREADY_DELETED");
39       RetCodeName[10] = new String("DDS_RETCODE_TIMEOUT");
40       RetCodeName[11] = new String("DDS_RETCODE_NO_DATA");
41         RetCodeName[12] = new String("DDS_RETCODE_ILLEGAL_OPERATION");
42    }
43
44    /**
45     * Returns the name of an error code.
46     **/
47    public static String getErrorName(int status) {
48       return RetCodeName[status];
49    }
50
51    /**
52     * Check the return status for errors. If there is an error,
53      * then terminate.
54     **/
55    public static void checkStatus(int status, String info) {
56       if ( status != RETCODE_OK.value &&
57             status != RETCODE_NO_DATA.value) {
58           System.out.println(
```

```
59                  "Error in " + info + ": " + getErrorName(status) );
60              System.exit(-1);
61          }
62      }
63
64      /**
65       * Check whether a valid handle has been returned. If not, then terminate.
66       **/
67      public static void checkHandle(Object handle, String info) {
68         if (handle == null) {
69              System.out.println(
70                   "Error in " + info + ": Creation failed: invalid handle");
71              System.exit(-1);
72           }
73      }
74
75  }
```

Chatter.java

```
1   /*********************************************************************
2    *
3    * Copyright (c) 2007
4    * PrismTech Ltd.
5    * All rights Reserved.
6    *
7    * LOGICAL_NAME:    Chatter.java
8    * FUNCTION:        OpenSplice DDS Tutorial example code.
9    * MODULE:          Tutorial for the Java programming language.
10   * DATE             june 2007.
11   *********************************************************************
12   *
13   * This file contains the implementation for the 'Chatter' executable.
14   *
15   ***/
16
17  package chatroom;
18
19  import DDS.*;
20  import Chat.*;
21
22  public class Chatter {
23
24      public static final int NUM_MSG = 10;
25      public static final int TERMINATION_MESSAGE = -1;
26
27      public static void main(String[] args) {
28          /* Generic DDS entities */
29          DomainParticipantFactory   dpf;
30          DomainParticipant          participant;
31          Topic                      chatMessageTopic;
32          Topic                      nameServiceTopic;
33          Publisher                  chatPublisher;
34          DataWriter                 parentWriter;
35
36          /* EntityQos holders */
37          TopicQosHolder         reliableTopicQos    = new TopicQosHolder();
38          TopicQosHolder         settingTopicQos     = new TopicQosHolder();
```

```
39          PublisherQosHolder     pubQos              = new PublisherQosHolder();
40          DataWriterQosHolder    dwQos               = new DataWriterQosHolder();
41
42          /* QosPolicy fields. */
43          WriterDataLifecycleQosPolicy writerDataLifecycle;
44
45          /* DDS Identifiers */
46          String                 domain = null;
47          long                   userHandle;
48          int                    status;
49
50          /* Type-specific DDS entities */
51          ChatMessageTypeSupport chatMessageTS;
52          NameServiceTypeSupport nameServiceTS;
53          ChatMessageDataWriter  talker;
54          NameServiceDataWriter  nameServer;
55
56          /* Sample definitions */
57          ChatMessage            msg                = new ChatMessage();
58          NameService            ns                 = new NameService();
59
60          /* Others */
61          int                    ownID = 1;
62          int                    i;
63          String                 chatterName = null;
64          String                 partitionName = new String("ChatRoom");
65           String                 chatMessageTypeName;
66           String                 nameServiceTypeName;
67
68
69          /* Options: Chatter [ownID [name]] */
70          if (args.length > 0) {
71              ownID = Integer.parseInt(args[0]);
72              if (args.length > 1) {
73                  chatterName = args[1];
74              }
75          }
76
77          /* Create a DomainParticipantFactory and a DomainParticipant
78              (using Default QoS settings. */
79          dpf = DomainParticipantFactory.get_instance ();
80          ErrorHandler.checkHandle(
81               dpf, "DDS.DomainParticipantFactory.get_instance");
82          participant = dpf.create_participant(
83               domain, PARTICIPANT_QOS_DEFAULT.value, null, ANY_STATUS.value);
84          ErrorHandler.checkHandle(
85               participant, "DDS.DomainParticipantFactory.create_participant");
86
87          /* Register the required datatype for ChatMessage. */
88          chatMessageTS = new ChatMessageTypeSupport();
89           ErrorHandler.checkHandle(
90               chatMessageTS, "new ChatMessageTypeSupport");
91           chatMessageTypeName = chatMessageTS.get_type_name();
92          status = chatMessageTS.register_type(
93               participant, chatMessageTypeName);
94          ErrorHandler.checkStatus(
95               status, "Chat.ChatMessageTypeSupport.register_type");
96
97          /* Register the required datatype for NameService. */
98          nameServiceTS = new NameServiceTypeSupport();
99           ErrorHandler.checkHandle(
```

```
100             nameServiceTS, "new NameServiceTypeSupport");
101         nameServiceTypeName = nameServiceTS.get_type_name();
102         status = nameServiceTS.register_type(
103             participant, nameServiceTypeName);
104         ErrorHandler.checkStatus(
105             status, "Chat.NameServiceTypeSupport.register_type");
106
107         /* Set the ReliabilityQosPolicy to RELIABLE. */
108         status = participant.get_default_topic_qos(reliableTopicQos);
109         ErrorHandler.checkStatus(
110             status, "DDS.DomainParticipant.get_default_topic_qos");
111         reliableTopicQos.value.reliability.kind =
112             ReliabilityQosPolicyKind.RELIABLE_RELIABILITY_QOS;
113
114         /* Make the tailored QoS the new default. */
115         status = participant.set_default_topic_qos(reliableTopicQos.value);
116         ErrorHandler.checkStatus(
117             status, "DDS.DomainParticipant.set_default_topic_qos");
118
119         /* Use the changed policy when defining the ChatMessage topic */
120         chatMessageTopic = participant.create_topic(
121             "Chat_ChatMessage",
122             chatMessageTypeName,
123             reliableTopicQos.value,
124             null,
125             ANY_STATUS.value);
126         ErrorHandler.checkHandle(
127         chatMessageTopic,
128         "DDS.DomainParticipant.create_topic (ChatMessage)");
129
130         /* Set the DurabilityQosPolicy to TRANSIENT. */
131         status = participant.get_default_topic_qos(settingTopicQos);
132         ErrorHandler.checkStatus(
133             status, "DDS.DomainParticipant.get_default_topic_qos");
134         settingTopicQos.value.durability.kind =
135             DurabilityQosPolicyKind.TRANSIENT_DURABILITY_QOS;
136
137         /* Create the NameService Topic. */
138         nameServiceTopic = participant.create_topic(
139             "Chat_NameService",
140             nameServiceTypeName,
141             settingTopicQos.value,
142             null,
143             ANY_STATUS.value);
144         ErrorHandler.checkHandle(
145             nameServiceTopic,
146             "DDS.DomainParticipant.create_topic (NameService)");
147
148         /* Adapt the default PublisherQos to write into the
149            "ChatRoom" Partition. */
150         status = participant.get_default_publisher_qos (pubQos);
151         ErrorHandler.checkStatus(
152             status, "DDS.DomainParticipant.get_default_publisher_qos");
153         pubQos.value.partition.name = new String[1];
154         pubQos.value.partition.name[0] = partitionName;
155
156         /* Create a Publisher for the chatter application. */
157         chatPublisher = participant.create_publisher(
158             pubQos.value, null, ANY_STATUS.value);
159         ErrorHandler.checkHandle(
160             chatPublisher, "DDS.DomainParticipant.create_publisher");
```

```
161
162         /* Create a DataWriter for the ChatMessage Topic
163            (using the appropriate QoS). */
164         parentWriter = chatPublisher.create_datawriter(
165             chatMessageTopic,
166             DATAWRITER_QOS_USE_TOPIC_QOS.value,
167             null,
168              ANY_STATUS.value);
169          ErrorHandler.checkHandle(
170              parentWriter, "DDS.Publisher.create_datawriter (chatMessage)");
171
172         /* Narrow the abstract parent into its typed representative. */
173         talker = ChatMessageDataWriterHelper.narrow(parentWriter);
174          ErrorHandler.checkHandle(
175              talker, "Chat.ChatMessageDataWriterHelper.narrow");
176
177          /* Create a DataWriter for the NameService Topic
178             (using the appropriate QoS). */
179         status = chatPublisher.get_default_datawriter_qos(dwQos);
180         ErrorHandler.checkStatus(
181             status, "DDS.Publisher.get_default_datawriter_qos");
182         status = chatPublisher.copy_from_topic_qos(
183             dwQos, settingTopicQos.value);
184         ErrorHandler.checkStatus(status, "DDS.Publisher.copy_from_topic_qos");
185         writerDataLifecycle = dwQos.value.writer_data_lifecycle;
186         writerDataLifecycle.autodispose_unregistered_instances = false;
187          parentWriter = chatPublisher.create_datawriter(
188              nameServiceTopic,
189              dwQos.value,
190              null,
191              ANY_STATUS.value);
192          ErrorHandler.checkHandle(
193              parentWriter, "DDS.Publisher.create_datawriter (NameService)");
194
195          /* Narrow the abstract parent into its typed representative. */
196          nameServer = NameServiceDataWriterHelper.narrow(parentWriter);
197          ErrorHandler.checkHandle(
198              nameServer, "Chat.NameServiceDataWriterHelper.narrow");
199
200          /* Initialize the NameServer attributes. */
201          ns.userID = ownID;
202          if (chatterName != null) {
203              ns.name = chatterName;
204          } else {
205              ns.name = "Chatter " + ownID;
206          }
207
208          /* Write the user-information into the system
209             (registering the instance implicitly). */
210          status = nameServer.write(ns, HANDLE_NIL.value);
211          ErrorHandler.checkStatus(status, "Chat.ChatMessageDataWriter.write");
212
213          /* Initialize the chat messages. */
214          msg.userID = ownID;
215          msg.index = 0;
216          if (ownID == TERMINATION_MESSAGE) {
217              msg.content = "Termination message.";
218          } else {
219              msg.content = "Hi there, I will send you " +
220                  NUM_MSG + " more messages.";
221          }
```

```
222         System.out.println("Writing message: \"" + msg.content + "\"");
223
224         /* Register a chat message for this user
225            (pre-allocating resources for it!!) */
226         userHandle = talker.register_instance(msg);
227
228         /* Write a message using the pre-generated instance handle. */
229         status = talker.write(msg, userHandle);
230         ErrorHandler.checkStatus(status, "Chat.ChatMessageDataWriter.write");
231
232         try {
233             Thread.sleep (1000); /* do not run so fast! */
234         } catch (InterruptedException e) {
235             e.printStackTrace();
236         }
237
238         /* Write any number of messages . */
239         for (i = 1; i <= NUM_MSG && ownID != TERMINATION_MESSAGE; i++) {
240             msg.index = i;
241             msg.content = "Message no. " + i;
242             System.out.println("Writing message: \"" + msg.content + "\"");
243             status = talker.write(msg, userHandle);
244             ErrorHandler.checkStatus(
245                 status, "Chat.ChatMessageDataWriter.write");
246             try {
247                 Thread.sleep (1000); /* do not run so fast! */
248             } catch (InterruptedException e) {
249                 e.printStackTrace();
250             }
251         }
252
253        /* Leave the room by disposing and unregistering the message instance */
254         status = talker.dispose(msg, userHandle);
255         ErrorHandler.checkStatus(
256             status, "Chat.ChatMessageDataWriter.dispose");
257         status = talker.unregister_instance(msg, userHandle);
258         ErrorHandler.checkStatus(
259             status, "Chat.ChatMessageDataWriter.unregister_instance");
260
261         /* Also unregister our name. */
262         status = nameServer.unregister_instance(ns, HANDLE_NIL.value);
263         ErrorHandler.checkStatus(
264             status, "Chat.NameServiceDataWriter.unregister_instance");
265
266         /* Remove the DataWriters */
267         status = chatPublisher.delete_datawriter(talker);
268         ErrorHandler.checkStatus(
269             status, "DDS.Publisher.delete_datawriter (talker)");
270
271         status = chatPublisher.delete_datawriter(nameServer);
272         ErrorHandler.checkStatus(status,
273             "DDS.Publisher.delete_datawriter (nameServer)");
274
275         /* Remove the Publisher. */
276         status = participant.delete_publisher(chatPublisher);
277         ErrorHandler.checkStatus(
278             status, "DDS.DomainParticipant.delete_publisher");
279
280         /* Remove the Topics. */
281         status = participant.delete_topic(nameServiceTopic);
282         ErrorHandler.checkStatus(
```

```
283             status, "DDS.DomainParticipant.delete_topic (nameServiceTopic)");
284
285         status = participant.delete_topic(chatMessageTopic);
286         ErrorHandler.checkStatus(
287             status, "DDS.DomainParticipant.delete_topic (chatMessageTopic)");
288
289         /* Remove the DomainParticipant. */
290         status = dpf.delete_participant(participant);
291         ErrorHandler.checkStatus(
292             status, "DDS.DomainParticipantFactory.delete_participant");
293     }
294 }
```

MessageBoard.java

```
1  /*********************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    MessageBoard.java
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the Java programming language.
10  * DATE             june 2007.
11  *********************************************************************
12  *
13  * This file contains the implementation for the 'MessageBoard' executable.
14  *
15  ***/
16
17 package chatroom;
18
19 import DDS.*;
20 import Chat.*;
21
22 public class MessageBoard {
23
24
25     public static final int TERMINATION_MESSAGE = -1;
26
27
28     public static void main(String[] args) {
29         /* Generic DDS entities */
30         DomainParticipantFactory  dpf;
31         DomainParticipant         parentDP;
32         ExtDomainParticipant      participant;
33         Topic                     chatMessageTopic;
34         Topic                     nameServiceTopic;
35         TopicDescription          namedMessageTopic;
36         Subscriber                chatSubscriber;
37         DataReader                parentReader;
38
39         /* Type-specific DDS entities */
40         ChatMessageTypeSupport  chatMessageTS;
41         NameServiceTypeSupport  nameServiceTS;
42         NamedMessageTypeSupport namedMessageTS;
43         NamedMessageDataReader  chatAdmin;
```

```
44          NamedMessageSeqHolder   msgSeq           = new NamedMessageSeqHolder();
45          SampleInfoSeqHolder     infoSeq           = new SampleInfoSeqHolder();
46
47          /* QosPolicy holders */
48          TopicQosHolder          reliableTopicQos = new TopicQosHolder();
49          TopicQosHolder          settingTopicQos  = new TopicQosHolder();
50          SubscriberQosHolder     subQos           = new SubscriberQosHolder();
51          String[]                parameterList;
52
53          /* DDS Identifiers */
54          String                  domain           = null;
55          int                     status;
56
57          /* Others */
58          boolean                 terminated       = false;
59          String                  partitionName    = new String("ChatRoom");
60          String                  chatMessageTypeName;
61          String                  nameServiceTypeName;
62          String                  namedMessageTypeName;
63
64          /* Options: MessageBoard [ownID] */
65          /* Messages having owner ownID will be ignored */
66          parameterList = new String[1];
67
68          if (args.length >0) {
69              parameterList[0] = args[0];
70          }
71          else
72          {
73              parameterList[0] = new String("0");
74          }
75
76          /* Create a DomainParticipantFactory and a DomainParticipant
77             (using Default QoS settings. */
78          dpf = DomainParticipantFactory.get_instance ();
79          ErrorHandler.checkHandle(
80              dpf, "DDS.DomainParticipantFactory.get_instance");
81          parentDP = dpf.create_participant(
82              domain, PARTICIPANT_QOS_DEFAULT.value, null, ANY_STATUS.value);
83          ErrorHandler.checkHandle(
84              parentDP, "DDS.DomainParticipantFactory.create_participant");
85
86          /* Register the required datatype for ChatMessage. */
87          chatMessageTS = new ChatMessageTypeSupport();
88          ErrorHandler.checkHandle(
89              chatMessageTS, "new ChatMessageTypeSupport");
90          chatMessageTypeName = chatMessageTS.get_type_name();
91          status = chatMessageTS.register_type(parentDP, chatMessageTypeName);
92          ErrorHandler.checkStatus(
93              status, "Chat.ChatMessageTypeSupport.register_type");
94
95          /* Register the required datatype for NameService. */
96          nameServiceTS = new NameServiceTypeSupport();
97          ErrorHandler.checkHandle(
98              nameServiceTS, "new NameServiceTypeSupport");
99          nameServiceTypeName = nameServiceTS.get_type_name();
100         nameServiceTS.register_type(parentDP, nameServiceTypeName);
101         ErrorHandler.checkStatus(
102             status, "Chat.NameServiceTypeSupport.register_type");
103
104         /* Register the required datatype for NamedMessage. */
```

```
105         namedMessageTS = new NamedMessageTypeSupport();
106         ErrorHandler.checkHandle(
107             namedMessageTS, "new NamedMessageTypeSupport");
108         namedMessageTypeName = namedMessageTS.get_type_name();
109         status = namedMessageTS.register_type(parentDP, namedMessageTypeName);
110         ErrorHandler.checkStatus(
111             status, "Chat.NamedMessageTypeSupport.register_type");
112
113         /* Narrow the normal participant to its extended representative */
114         participant = ExtDomainParticipantHelper.narrow(parentDP);
115         ErrorHandler.checkHandle(
116             participant, "ExtDomainParticipantHelper.narrow");
117
118         /* Set the ReliabilityQosPolicy to RELIABLE. */
119         status = participant.get_default_topic_qos(reliableTopicQos);
120         ErrorHandler.checkStatus(
121             status, "DDS.DomainParticipant.get_default_topic_qos");
122         reliableTopicQos.value.reliability.kind =
123             ReliabilityQosPolicyKind.RELIABLE_RELIABILITY_QOS;
124
125         /* Make the tailored QoS the new default. */
126         status = participant.set_default_topic_qos(reliableTopicQos.value);
127         ErrorHandler.checkStatus(
128             status, "DDS.DomainParticipant.set_default_topic_qos");
129
130         /* Use the changed policy when defining the ChatMessage topic */
131         chatMessageTopic = participant.create_topic(
132             "Chat_ChatMessage",
133             chatMessageTypeName,
134             reliableTopicQos.value,
135             null,
136             ANY_STATUS.value);
137         ErrorHandler.checkHandle(
138             chatMessageTopic,
139             "DDS.DomainParticipant.create_topic (ChatMessage)");
140
141         /* Set the DurabilityQosPolicy to TRANSIENT. */
142         status = participant.get_default_topic_qos(settingTopicQos);
143         ErrorHandler.checkStatus(
144             status, "DDS.DomainParticipant.get_default_topic_qos");
145         settingTopicQos.value.durability.kind =
146             DurabilityQosPolicyKind.TRANSIENT_DURABILITY_QOS;
147
148         /* Create the NameService Topic. */
149         nameServiceTopic = participant.create_topic(
150             "Chat_NameService",
151             nameServiceTypeName,
152             settingTopicQos.value,
153             null,
154             ANY_STATUS.value);
155         ErrorHandler.checkHandle(
156             nameServiceTopic,
157             "DDS.DomainParticipant.create_topic (NameService)");
158
159         /* Create a multitopic that substitutes the userID
160            with its corresponding userName. */
161         namedMessageTopic = participant.create_simulated_multitopic(
162             "Chat_NamedMessage",
163             namedMessageTypeName,
164             "SELECT userID, name AS userName, index, content " +
165                 "FROM Chat_NameService NATURAL JOIN Chat_ChatMessage " +
```

```
166                 "WHERE userID <> %0",
167             parameterList);
168         ErrorHandler.checkHandle(
169             namedMessageTopic,
170             "ExtDomainParticipant.create_simulated_multitopic");
171
172         /* Adapt the default SubscriberQos to read from the
173            "ChatRoom" Partition. */
174         status = participant.get_default_subscriber_qos (subQos);
175         ErrorHandler.checkStatus(
176             status, "DDS.DomainParticipant.get_default_subscriber_qos");
177         subQos.value.partition.name = new String[1];
178         subQos.value.partition.name[0] = partitionName;
179
180         /* Create a Subscriber for the MessageBoard application. */
181         chatSubscriber = participant.create_subscriber(
182             subQos.value, null, ANY_STATUS.value);
183         ErrorHandler.checkHandle(
184             chatSubscriber, "DDS.DomainParticipant.create_subscriber");
185
186         /* Create a DataReader for the NamedMessage Topic
187            (using the appropriate QoS). */
188         parentReader = chatSubscriber.create_datareader(
189             namedMessageTopic,
190             DATAREADER_QOS_USE_TOPIC_QOS.value,
191             null,
192             ANY_STATUS.value);
193         ErrorHandler.checkHandle(
194             parentReader, "DDS.Subscriber.create_datareader");
195
196         /* Narrow the abstract parent into its typed representative. */
197         chatAdmin = NamedMessageDataReaderHelper.narrow(parentReader);
198         ErrorHandler.checkHandle(
199             chatAdmin, "Chat.NamedMessageDataReaderHelper.narrow");
200
201         /* Print a message that the MessageBoard has opened. */
202         System.out.println(
203                 "MessageBoard has opened: send a ChatMessage " +
204                 "with userID = -1 to close it....\n");
205
206         while (!terminated) {
207             /* Note: using read does not remove the samples from
208                unregistered instances from the DataReader. This means
209                that the DataRase would use more and more resources.
210                That's why we use take here instead. */
211
212             status = chatAdmin.take(
213                 msgSeq,
214                 infoSeq,
215                 LENGTH_UNLIMITED.value,
216                 ANY_SAMPLE_STATE.value,
217                 ANY_VIEW_STATE.value,
218                 ALIVE_INSTANCE_STATE.value );
219             ErrorHandler.checkStatus(
220                 status, "Chat.NamedMessageDataReader.take");
221
222             for (int i = 0; i < msgSeq.value.length; i++) {
223                 if (msgSeq.value[i].userID == TERMINATION_MESSAGE) {
224                     System.out.println(
225                         "Termination message received: exiting...");
226                     terminated = true;
```

```
227                 } else {
228                     System.out.println(
229                         msgSeq.value[i].userName + ": " +
230                         msgSeq.value[i].content);
231                 }
232             }
233
234             status = chatAdmin.return_loan(msgSeq, infoSeq);
235             ErrorHandler.checkStatus(
236                 status, "Chat.ChatMessageDataReader.return_loan");
237
238             msgSeq.value = null;
239             infoSeq.value = null;
240
241             /* Sleep for some amount of time, as not to consume
242                too much CPU cycles. */
243             try {
244                 Thread.sleep(100);
245             } catch (InterruptedException e) {
246                 e.printStackTrace();
247             }
248         }
249
250         /* Remove the DataReader */
251         status = chatSubscriber.delete_datareader(chatAdmin);
252         ErrorHandler.checkStatus(
253             status, "DDS.Subscriber.delete_datareader");
254
255         /* Remove the Subscriber. */
256         status = participant.delete_subscriber(chatSubscriber);
257         ErrorHandler.checkStatus(
258             status, "DDS.DomainParticipant.delete_subscriber");
259
260         /* Remove the Topics. */
261         status = participant.delete_simulated_multitopic(namedMessageTopic);
262         ErrorHandler.checkStatus(
263             status, "DDS.ExtDomainParticipant.delete_simulated_multitopic");
264
265         status = participant.delete_topic(nameServiceTopic);
266         ErrorHandler.checkStatus(
267             status, "DDS.DomainParticipant.delete_topic (nameServiceTopic)");
268
269         status = participant.delete_topic(chatMessageTopic);
270         ErrorHandler.checkStatus(
271             status, "DDS.DomainParticipant.delete_topic (chatMessageTopic)");
272
273         /* Remove the DomainParticipant. */
274         status = dpf.delete_participant(parentDP);
275         ErrorHandler.checkStatus(
276             status, "DDS.DomainParticipantFactory.delete_participant");
277     }
278
279 }
```

DataReaderListenerImpl.java

```
1  /*********************************************************************
2   *
```

```
3    * Copyright (c) 2007
4    * PrismTech Ltd.
5    * All rights Reserved.
6    *
7    * LOGICAL_NAME:    DataReaderListenerImpl.java
8    * FUNCTION:        OpenSplice DDS Tutorial example code.
9    * MODULE:          Tutorial for the Java programming language.
10   * DATE             june 2007.
11   ********************************************************************
12   *
13   * This file contains the implementation for a DataReader listener, that
14   * simulates MultiTopic behavior by writing a NamedMessage sample (which
15   * contains the merged information from both the ChatMessage and NameService
16   * topics) for each incoming ChatMessage.
17   *
18   ***/
19
20 package chatroom;
21
22 import DDS.*;
23 import Chat.*;
24
25 public class DataReaderListenerImpl implements DataReaderListener {
26
27     /***
28      * Attributes
29      ***/
30     /* Caching variables */
31     private int                   previous      = 0x80000000;
32     private String                userName;
33     private ChatMessageSeqHolder  msgSeq        = new ChatMessageSeqHolder();
34     private NameServiceSeqHolder  nameSeq       = new NameServiceSeqHolder();
35     private SampleInfoSeqHolder   infoSeq1      = new SampleInfoSeqHolder();
36     private SampleInfoSeqHolder   infoSeq2      = new SampleInfoSeqHolder();
37     private NamedMessage          joinedSample  = new NamedMessage();
38
39
40     /* Type-specific DDS entities */
41     public ChatMessageDataReader        chatMessageDR;
42     public NameServiceDataReader        nameServiceDR;
43     public NamedMessageDataWriter       namedMessageDW;
44
45     /* Query related stuff */
46     public QueryCondition               nameFinder;
47     public String[]                     nameFinderParams;
48
49     /***
50      * Operations
51      ***/
52     public void on_requested_deadline_missed(
53             DataReader the_reader,
54             RequestedDeadlineMissedStatus status) { }
55
56     public void on_requested_incompatible_qos(
57             DataReader the_reader,
58             RequestedIncompatibleQosStatus status) { }
59
60     public void on_sample_rejected(
61         DataReader the_reader, SampleRejectedStatus status) { }
62
63     public void on_liveliness_changed(
```

```
64          DataReader the_reader, LivelinessChangedStatus status) { }
65
66      public void on_data_available(DataReader the_reader) {
67
68          /* Take all messages. */
69          int status = chatMessageDR.take(
70              msgSeq,
71              infoSeq1,
72              LENGTH_UNLIMITED.value,
73              ANY_SAMPLE_STATE.value,
74              ANY_VIEW_STATE.value,
75              ANY_INSTANCE_STATE.value);
76          ErrorHandler.checkStatus(
77              status, "Chat.ChatMessageDataReader.take");
78
79          /* For each message, extract the key-field and find
80             the corresponding name. */
81          for (int i = 0; i < msgSeq.value.length; i++)
82          {
83              if (infoSeq1.value[i].valid_data)
84              {
85                  /* Find the corresponding named message. */
86                  if (msgSeq.value[i].userID != previous)
87                  {
88                      previous = msgSeq.value[i].userID;
89                      nameFinderParams[0] = Integer.toString(previous);
90                     status = nameFinder.set_query_parameters(nameFinderParams);
91                      ErrorHandler.checkStatus(
92                          status, "DDS.QueryCondition.set_query_parameters");
93                      status = nameServiceDR.read_w_condition(
94                          nameSeq,
95                          infoSeq2,
96                          LENGTH_UNLIMITED.value,
97                          nameFinder);
98                      ErrorHandler.checkStatus(
99                         status, "Chat.NameServiceDataReader.read_w_condition");
100
101                     /* Extract Name (there should only be one result). */
102                     if (status == RETCODE_NO_DATA.value)
103                     {
104                         userName = new String(
105                                 "Name not found!! id = " + previous);
106                     }
107                     else
108                     {
109                         userName = nameSeq.value[0].name;
110                     }
111
112                     /* Release the name sample again. */
113                     status = nameServiceDR.return_loan(nameSeq, infoSeq2);
114                     ErrorHandler.checkStatus(
115                         status, "Chat.NameServiceDataReader.return_loan");
116                 }
117                 /* Write merged Topic with userName instead of userID. */
118                 joinedSample.userName = userName;
119                 joinedSample.userID = msgSeq.value[i].userID;
120                 joinedSample.index = msgSeq.value[i].index;
121                 joinedSample.content = msgSeq.value[i].content;
122                 status = namedMessageDW.write(joinedSample, HANDLE_NIL.value);
123                 ErrorHandler.checkStatus(
124                     status, "Chat.NamedMessageDataWriter.write");
```

```
125             }
126         }
127         status = chatMessageDR.return_loan(msgSeq, infoSeq1);
128         ErrorHandler.checkStatus(
129             status, "Chat.ChatMessageDataReader.return_loan");
130
131     }
132
133     public void on_subscription_matched(
134         DataReader the_reader, SubscriptionMatchedStatus status) { }
135
136     public void on_sample_lost(
137         DataReader the_reader, SampleLostStatus status) { }
138
139 }
```

ExtDomainParticipant.java

```
1  /*************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    ExtDomainParticipant.java
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the Java programming language.
10  * DATE             june 2007.
11  *************************************************************************
12  *
13  * This file contains the implementation for an extended DomainParticipant
14  * class, that adds a new operations named 'simulate_multitopic', which
15  * simulates the behavior of a multitopic by combining a ContentFilteredTopic
16  * with a QueryCondition and a DataReaderListener.
17  *
18  ***/
19
20 package chatroom;
21
22 import DDS.*;
23 import Chat.*;
24
25 public class ExtDomainParticipant implements DomainParticipant {
26
27     /***
28      * Attributes
29      ***/
30
31     // Encapsulated DomainParticipant.
32     private DomainParticipant               realParticipant;
33
34     /*Implementation for DataReaderListener */
35     private DataReaderListenerImpl          msgListener;
36
37     /* Generic DDS entities */
38     private Topic                           chatMessageTopic;
39     private Topic                           nameServiceTopic;
40     private ContentFilteredTopic            filteredMessageTopic;
```

```
41      private Topic                           namedMessageTopic;
42      private Subscriber                      multiSub;
43      private Publisher                       multiPub;
44
45
46      /***
47       * Constructor
48       ***/
49      ExtDomainParticipant(DomainParticipant aParticipant) {
50          this.realParticipant = aParticipant;
51      }
52
53
54      /***
55       * Operations
56       ***/
57      public Topic create_simulated_multitopic (
58          String name,
59          String type_name,
60          String subscription_expression,
61          String[] expression_parameters)
62      {
63
64          /* Type-specific DDS entities */
65          ChatMessageDataReader   chatMessageDR;
66          NameServiceDataReader   nameServiceDR;
67          NamedMessageDataWriter  namedMessageDW;
68
69          /* Query related stuff */
70          QueryCondition          nameFinder;
71          String[]                nameFinderParams;
72
73          /* QosPolicy holders */
74          TopicQosHolder          namedMessageQos = new TopicQosHolder();
75         SubscriberQosHolder     subQos          = new SubscriberQosHolder();
76          PublisherQosHolder      pubQos          = new PublisherQosHolder();
77
78          /* Others */
79          DataReader              parentReader;
80          DataWriter              parentWriter;
81          String                  partitionName   = new String("ChatRoom");
82          String                  nameFinderExpr;
83          int                     status;
84
85          /* Lookup both components that constitute the multi-topic. */
86          chatMessageTopic = realParticipant.find_topic(
87              "Chat_ChatMessage", DURATION_INFINITY.value);
88          ErrorHandler.checkHandle(
89              chatMessageTopic,
90              "DDS.DomainParticipant.find_topic (Chat_ChatMessage)");
91
92          nameServiceTopic = realParticipant.find_topic(
93              "Chat_NameService", DURATION_INFINITY.value);
94          ErrorHandler.checkHandle(
95              nameServiceTopic,
96              "DDS.DomainParticipant.find_topic (Chat_NameService)");
97
98          /* Create a ContentFilteredTopic to filter out
99             our own ChatMessages. */
100         filteredMessageTopic = realParticipant.create_contentfilteredtopic(
101             "Chat_FilteredMessage",
```

```
102             chatMessageTopic,
103             "userID <> %0",
104             expression_parameters);
105         ErrorHandler.checkHandle(
106             filteredMessageTopic,
107             "DDS.DomainParticipant.create_contentfilteredtopic");
108
109
110         /* Adapt the default SubscriberQos to read from the
111            "ChatRoom" Partition. */
112         status = realParticipant.get_default_subscriber_qos (subQos);
113         ErrorHandler.checkStatus(
114             status, "DDS.DomainParticipant.get_default_subscriber_qos");
115         subQos.value.partition.name = new String[1];
116         subQos.value.partition.name[0] = partitionName;
117
118         /* Create a private Subscriber for the multitopic simulator. */
119         multiSub = realParticipant.create_subscriber(
120             subQos.value, null, ANY_STATUS.value);
121         ErrorHandler.checkHandle(
122             multiSub,
123             "DDS.DomainParticipant.create_subscriber (for multitopic)");
124
125         /* Create a DataReader for the FilteredMessage Topic
126            (using the appropriate QoS). */
127         parentReader = multiSub.create_datareader(
128             filteredMessageTopic,
129             DATAREADER_QOS_USE_TOPIC_QOS.value,
130             null,
131             ANY_STATUS.value);
132         ErrorHandler.checkHandle(
133             parentReader, "DDS.Subscriber.create_datareader (ChatMessage)");
134
135         /* Narrow the abstract parent into its typed representative. */
136         chatMessageDR = ChatMessageDataReaderHelper.narrow(parentReader);
137         ErrorHandler.checkHandle(
138             chatMessageDR, "Chat.ChatMessageDataReaderHelper.narrow");
139
140         /* Allocate the DataReaderListener Implementation. */
141         msgListener = new DataReaderListenerImpl();
142         ErrorHandler.checkHandle(msgListener, "new DataReaderListenerImpl");
143
144         /* Attach the DataReaderListener to the DataReader,
145            only enabling the data_available event. */
146         status = chatMessageDR.set_listener(
147             msgListener, DDS.DATA_AVAILABLE_STATUS.value);
148         ErrorHandler.checkStatus(status, "DDS.DataReader_set_listener");
149
150         /* Create a DataReader for the nameService Topic
151            (using the appropriate QoS). */
152         parentReader = multiSub.create_datareader(
153             nameServiceTopic,
154             DATAREADER_QOS_USE_TOPIC_QOS.value,
155             null,
156             ANY_STATUS.value);
157         ErrorHandler.checkHandle(
158             parentReader, "DDS.Subscriber.create_datareader (NameService)");
159
160         /* Narrow the abstract parent into its typed representative. */
161         nameServiceDR = NameServiceDataReaderHelper.narrow(parentReader);
162         ErrorHandler.checkHandle(
```

```
163             nameServiceDR, "Chat.NameServiceDataReaderHelper.narrow");
164
165         /* Define the SQL expression (using a parameterized value). */
166         nameFinderExpr = new String("userID = %0");
167
168         /* Allocate and assign the query parameters. */
169         nameFinderParams = new String[1];
170         nameFinderParams[0] = expression_parameters[0];
171
172         /* Create a QueryCondition to only read corresponding
173            nameService information by key-value. */
174         nameFinder = nameServiceDR.create_querycondition(
175             ANY_SAMPLE_STATE.value,
176             ANY_VIEW_STATE.value,
177             ANY_INSTANCE_STATE.value,
178             nameFinderExpr,
179             nameFinderParams);
180         ErrorHandler.checkHandle(
181             nameFinder, "DDS.DataReader.create_querycondition (nameFinder)");
182
183         /* Create the Topic that simulates the multi-topic
184            (use Qos from chatMessage).*/
185         status = chatMessageTopic.get_qos(namedMessageQos);
186         ErrorHandler.checkStatus(status, "DDS.Topic.get_qos");
187
188         /* Create the NamedMessage Topic whose samples simulate
189            the MultiTopic */
190         namedMessageTopic = realParticipant.create_topic(
191             "Chat_NamedMessage",
192             type_name,
193             namedMessageQos.value,
194             null,
195             ANY_STATUS.value);
196         ErrorHandler.checkHandle(
197             namedMessageTopic,
198             "DDS.DomainParticipant.create_topic (NamedMessage)");
199
200         /* Adapt the default PublisherQos to write into the
201            "ChatRoom" Partition. */
202         status = realParticipant.get_default_publisher_qos(pubQos);
203         ErrorHandler.checkStatus(
204             status, "DDS.DomainParticipant.get_default_publisher_qos");
205         pubQos.value.partition.name = new String[1];
206         pubQos.value.partition.name[0] = partitionName;
207
208         /* Create a private Publisher for the multitopic simulator. */
209         multiPub = realParticipant.create_publisher(
210             pubQos.value, null, ANY_STATUS.value);
211         ErrorHandler.checkHandle(
212             multiPub,
213             "DDS.DomainParticipant.create_publisher (for multitopic)");
214
215         /* Create a DataWriter for the multitopic. */
216         parentWriter = multiPub.create_datawriter(
217             namedMessageTopic,
218             DATAWRITER_QOS_USE_TOPIC_QOS.value,
219             null,
220             ANY_STATUS.value);
221         ErrorHandler.checkHandle(
222             parentWriter, "DDS.Publisher.create_datawriter (NamedMessage)");
223
```

```
224         /* Narrow the abstract parent into its typed representative. */
225         namedMessageDW = NamedMessageDataWriterHelper.narrow(parentWriter);
226         ErrorHandler.checkHandle(
227             namedMessageDW, "Chat.NamedMessageDataWriterHelper.narrow");
228
229         /* Store the relevant Entities in our Listener. */
230         msgListener.chatMessageDR = chatMessageDR;
231         msgListener.nameServiceDR = nameServiceDR;
232         msgListener.namedMessageDW = namedMessageDW;
233         msgListener.nameFinder = nameFinder;
234         msgListener.nameFinderParams = nameFinderParams;
235
236         /* Return the simulated Multitopic. */
237         return namedMessageTopic;
238
239     }
240
241     public int delete_simulated_multitopic(
242         TopicDescription smt)
243     {
244         int status;
245
246         /* Remove the DataWriter */
247         status = multiPub.delete_datawriter(msgListener.namedMessageDW);
248         ErrorHandler.checkStatus(status, "DDS.Publisher.delete_datawriter");
249
250         /* Remove the Publisher. */
251         status = realParticipant.delete_publisher(multiPub);
252         ErrorHandler.checkStatus(
253             status, "DDS.DomainParticipant.delete_publisher");
254
255         /* Remove the QueryCondition. */
256         status = msgListener.nameServiceDR.delete_readcondition(
257             msgListener.nameFinder);
258         ErrorHandler.checkStatus(
259             status, "DDS.DataReader.delete_readcondition");
260
261         /* Remove the DataReaders. */
262         status = multiSub.delete_datareader(msgListener.nameServiceDR);
263         ErrorHandler.checkStatus(status, "DDS.Subscriber.delete_datareader");
264         status = multiSub.delete_datareader(msgListener.chatMessageDR);
265         ErrorHandler.checkStatus(status, "DDS.Subscriber.delete_datareader");
266
267         /* Remove the Subscriber. */
268         status = realParticipant.delete_subscriber(multiSub);
269         ErrorHandler.checkStatus(
270             status, "DDS.DomainParticipant.delete_subscriber");
271
272         /* Remove the ContentFilteredTopic. */
273         status = realParticipant.delete_contentfilteredtopic(
274             filteredMessageTopic);
275         ErrorHandler.checkStatus(
276             status, "DDS.DomainParticipant.delete_contentfilteredtopic");
277
278         /* Remove all other topics. */
279         status = realParticipant.delete_topic(namedMessageTopic);
280         ErrorHandler.checkStatus(
281             status, "DDS.DomainParticipant.delete_topic (namedMessageTopic)");
282         status = realParticipant.delete_topic(nameServiceTopic);
283         ErrorHandler.checkStatus(
284             status, "DDS.DomainParticipant.delete_topic (nameServiceTopic)");
```

```
285         status = realParticipant.delete_topic(chatMessageTopic);
286         ErrorHandler.checkStatus(
287             status, "DDS.DomainParticipant.delete_topic (chatMessageTopic)");
288 
289         return status;
290     };
291 
292     public Publisher create_publisher(
293             PublisherQos qos, PublisherListener a_listener, int mask) {
294         return realParticipant.create_publisher(qos, a_listener, mask);
295     }
296 
297     public int delete_publisher(Publisher p) {
298         return realParticipant.delete_publisher(p);
299     }
300 
301     public Subscriber create_subscriber(
302             SubscriberQos qos, SubscriberListener a_listener, int mask) {
303         return realParticipant.create_subscriber(qos, a_listener, mask);
304     }
305 
306     public int delete_subscriber(Subscriber s) {
307         return realParticipant.delete_subscriber(s);
308     }
309 
310     public Subscriber get_builtin_subscriber() {
311         return realParticipant.get_builtin_subscriber();
312     }
313 
314     public Topic create_topic(
315             String topic_name,
316             String type_name,
317             TopicQos qos,
318             TopicListener a_listener,
319             int mask) {
320         return realParticipant.create_topic(
321             topic_name, type_name, qos, a_listener, mask);
322     }
323 
324     public int delete_topic(Topic a_topic) {
325         return realParticipant.delete_topic(a_topic);
326     }
327 
328     public Topic find_topic(String topic_name, Duration_t timeout) {
329         return realParticipant.find_topic(topic_name, timeout);
330     }
331 
332     public TopicDescription lookup_topicdescription(String name) {
333         return realParticipant.lookup_topicdescription(name);
334     }
335 
336     public ContentFilteredTopic create_contentfilteredtopic(
337             String name,
338             Topic related_topic,
339             String filter_expression,
340             String[] filter_parameters) {
341         return realParticipant.create_contentfilteredtopic(
342                 name,
343                 related_topic,
344                 filter_expression,
345                 filter_parameters);
```

```
346     }
347
348     public int delete_contentfilteredtopic(
349             ContentFilteredTopic a_contentfilteredtopic) {
350         return realParticipant.delete_contentfilteredtopic(
351                 a_contentfilteredtopic);
352     }
353
354     public MultiTopic create_multitopic(
355             String name,
356             String type_name,
357             String subscription_expression,
358             String[] expression_parameters) {
359         return realParticipant.create_multitopic(
360                 name,
361                 type_name,
362                 subscription_expression,
363                 expression_parameters);
364     }
365
366     public int delete_multitopic(MultiTopic a_multitopic) {
367         return realParticipant.delete_multitopic(a_multitopic);
368     }
369
370     public int delete_contained_entities() {
371         return realParticipant.delete_contained_entities();
372     }
373
374     public int set_qos(DomainParticipantQos qos) {
375         return realParticipant.set_qos(qos);
376     }
377
378     public int get_qos(DomainParticipantQosHolder qos) {
379         return realParticipant.get_qos(qos);
380     }
381
382     public int set_listener(DomainParticipantListener a_listener, int mask) {
383         return realParticipant.set_listener(a_listener, mask);
384     }
385
386     public DomainParticipantListener get_listener() {
387         return realParticipant.get_listener();
388     }
389
390     public int ignore_participant(long handle) {
391         return realParticipant.ignore_participant(handle);
392     }
393
394     public int ignore_topic(long handle) {
395         return realParticipant.ignore_topic(handle);
396     }
397
398     public int ignore_publication(long handle) {
399         return realParticipant.ignore_publication(handle);
400     }
401
402     public int ignore_subscription(long handle) {
403         return realParticipant.ignore_subscription(handle);
404     }
405
406     public String get_domain_id() {
```

```
407         return realParticipant.get_domain_id();
408     }
409
410     public int assert_liveliness() {
411         return realParticipant.assert_liveliness();
412     }
413
414     public int set_default_publisher_qos(PublisherQos qos) {
415         return realParticipant.set_default_publisher_qos(qos);
416     }
417
418     public int get_default_publisher_qos(PublisherQosHolder qos) {
419         return realParticipant.get_default_publisher_qos(qos);
420     }
421
422     public int set_default_subscriber_qos(SubscriberQos qos) {
423         return realParticipant.set_default_subscriber_qos(qos);
424     }
425
426     public int get_default_subscriber_qos(SubscriberQosHolder qos) {
427         return realParticipant.get_default_subscriber_qos(qos);
428     }
429
430     public int set_default_topic_qos(TopicQos qos) {
431         return realParticipant.set_default_topic_qos(qos);
432     }
433
434     public int get_default_topic_qos(TopicQosHolder qos) {
435         return realParticipant.get_default_topic_qos(qos);
436     }
437
438     public int get_discovered_participants(InstanceHandleSeqHolder handles) {
439         return realParticipant.get_discovered_participants(handles);
440     }
441
442     public int get_discovered_participant_data(
443             long participant_handle,
444             ParticipantBuiltinTopicDataHolder participant_data) {
445         return realParticipant.get_discovered_participant_data(
446                 participant_handle, participant_data);
447     }
448
449     public int get_discovered_topics(InstanceHandleSeqHolder handles) {
450         return realParticipant.get_discovered_topics(handles);
451     }
452
453     public int get_discovered_topic_data(
454             long topic_handle,
455             TopicBuiltinTopicDataHolder topic_data) {
456         return realParticipant.get_discovered_topic_data(
457                 topic_handle, topic_data);
458     }
459
460     public boolean contains_entity(long a_handle) {
461         return realParticipant.contains_entity(a_handle);
462     }
463
464     public int get_current_time(Time_tHolder current_time) {
465         return realParticipant.get_current_time(current_time);
466     }
467
```

```
468     public int enable() {
469         return realParticipant.enable();
470     }
471
472     public StatusCondition get_statuscondition() {
473         return realParticipant.get_statuscondition();
474     }
475
476     public int get_status_changes() {
477         return realParticipant.get_status_changes();
478     }
479
480     public long get_instance_handle() {
481         return realParticipant.get_instance_handle();
482     }
483
484 }
```

ExtDomainParticipantHelper.java

```
1  /*************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
5   * All rights Reserved.
6   *
7   * LOGICAL_NAME:    ExtDomainParticipantHelper.java
8   * FUNCTION:        OpenSplice DDS Tutorial example code.
9   * MODULE:          Tutorial for the Java programming language.
10  * DATE             june 2007.
11  *************************************************************************
12  *
13  * This file contains the implementation for a Helper class of the extended
14  * DomainParticipant, that simulates the behavior of a Helper class with respect
15  * to narrowing an existing DomainParticipant into its extended representation.
16  *
17  ***/
18
19 package chatroom;
20
21 import DDS.DomainParticipant;
22
23 public class ExtDomainParticipantHelper {
24     public static ExtDomainParticipant narrow(
25             DomainParticipant participant) {
26         return new ExtDomainParticipant(participant);
27     }
28 }
```

UserLoad.java

```
1  /*************************************************************************
2   *
3   * Copyright (c) 2007
4   * PrismTech Ltd.
```

```
5    * All rights Reserved.
6    *
7    * LOGICAL_NAME:    UserLoad.java
8    * FUNCTION:        OpenSplice DDS Tutorial example code.
9    * MODULE:          Tutorial for the Java programming language.
10   * DATE             june 2007.
11   ************************************************************************
12   *
13   * This file contains the implementation for the 'UserLoad' executable.
14   *
15   ***/
16
17 package chatroom;
18
19 import DDS.*;
20 import Chat.*;
21
22 public class UserLoad extends Thread {
23
24     /* entities required by all threads. */
25     public static GuardCondition            escape;
26
27     /**
28      * Sleeper thread: sleeps 60 seconds and then triggers the WaitSet.
29      */
30     public void run() {
31         int status;
32
33         try {
34             sleep(60000);
35         } catch (InterruptedException e) {
36             e.printStackTrace();
37         }
38         status = escape.set_trigger_value(true);
39         ErrorHandler.checkStatus(
40             status, "DDS.GuardCondition.set_trigger_value");
41     }
42
43     public static void main(String[] args) {
44         /* Generic DDS entities */
45         DomainParticipant               participant;
46         Topic                           chatMessageTopic;
47         Topic                           nameServiceTopic;
48         Subscriber                      chatSubscriber;
49         DataReader                      parentReader;
50         QueryCondition                  singleUser;
51         ReadCondition                   newUser;
52         StatusCondition                 leftUser;
53         WaitSet                         userLoadWS;
54         LivelinessChangedStatusHolder   livChangStatus =
55             new LivelinessChangedStatusHolder();
56
57         /* QosPolicy holders */
58         TopicQosHolder          settingTopicQos  = new TopicQosHolder();
59         TopicQosHolder          reliableTopicQos = new TopicQosHolder();
60         SubscriberQosHolder     subQos           = new SubscriberQosHolder();
61         DataReaderQosHolder     messageQos       = new DataReaderQosHolder();
62
63         /* DDS Identifiers */
64         String                  domain           = null;
65         int                     status;
```

```
66          ConditionSeqHolder      guardList        = new ConditionSeqHolder();
67
68          /* Type-specific DDS entities */
69          ChatMessageTypeSupport  chatMessageTS;
70          NameServiceTypeSupport  nameServiceTS;
71          NameServiceDataReader   nameServer;
72          ChatMessageDataReader   loadAdmin;
73          ChatMessageSeqHolder    msgList          = new ChatMessageSeqHolder();
74          NameServiceSeqHolder    nsList           = new NameServiceSeqHolder();
75          SampleInfoSeqHolder     infoSeq          = new SampleInfoSeqHolder();
76          SampleInfoSeqHolder     infoSeq2         = new SampleInfoSeqHolder();
77
78          /* Others */
79          String[]                params;
80          String                  chatMessageTypeName;
81          String                  nameServiceTypeName;
82          boolean                 closed           = false;
83          int                     prevCount        = 0;
84
85          /* Create a DomainParticipant (using the
86             'TheParticipantFactory' convenience macro). */
87          participant = TheParticipantFactory.value.create_participant (
88              domain,
89              PARTICIPANT_QOS_DEFAULT.value,
90              null,
91              ANY_STATUS.value);
92          ErrorHandler.checkHandle(
93              participant, "DDS.DomainParticipantFactory.create_participant");
94
95          /* Register the required datatype for ChatMessage. */
96          chatMessageTS = new ChatMessageTypeSupport();
97          ErrorHandler.checkHandle(
98              chatMessageTS, "new ChatMessageTypeSupport");
99          chatMessageTypeName = chatMessageTS.get_type_name();
100         status = chatMessageTS.register_type(
101             participant, chatMessageTypeName);
102         ErrorHandler.checkStatus(
103             status, "Chat.ChatMessageTypeSupport.register_type");
104
105         /* Register the required datatype for NameService. */
106         nameServiceTS = new NameServiceTypeSupport();
107         ErrorHandler.checkHandle(
108             nameServiceTS, "new NameServiceTypeSupport");
109         nameServiceTypeName = nameServiceTS.get_type_name();
110         status = nameServiceTS.register_type(
111             participant, nameServiceTypeName);
112         ErrorHandler.checkStatus(
113             status, "Chat.NameServiceTypeSupport.register_type");
114
115         /* Set the ReliabilityQosPolicy to RELIABLE. */
116         status = participant.get_default_topic_qos(reliableTopicQos);
117         ErrorHandler.checkStatus(
118             status, "DDS.DomainParticipant.get_default_topic_qos");
119         reliableTopicQos.value.reliability.kind =
120             ReliabilityQosPolicyKind.RELIABLE_RELIABILITY_QOS;
121
122         /* Make the tailored QoS the new default. */
123         status = participant.set_default_topic_qos(reliableTopicQos.value);
124         ErrorHandler.checkStatus(
125             status, "DDS.DomainParticipant.set_default_topic_qos");
126
```

```
127         /* Use the changed policy when defining the ChatMessage topic */
128         chatMessageTopic = participant.create_topic(
129             "Chat_ChatMessage",
130             chatMessageTypeName,
131             reliableTopicQos.value,
132             null,
133             ANY_STATUS.value);
134         ErrorHandler.checkHandle(
135             chatMessageTopic,
136             "DDS.DomainParticipant.create_topic (ChatMessage)");
137
138         /* Set the DurabilityQosPolicy to TRANSIENT. */
139         status = participant.get_default_topic_qos(settingTopicQos);
140         ErrorHandler.checkStatus(
141             status, "DDS.DomainParticipant.get_default_topic_qos");
142         settingTopicQos.value.durability.kind =
143             DurabilityQosPolicyKind.TRANSIENT_DURABILITY_QOS;
144
145         /* Create the NameService Topic. */
146         nameServiceTopic = participant.create_topic(
147             "Chat_NameService",
148             nameServiceTypeName,
149             settingTopicQos.value,
150             null,
151             ANY_STATUS.value);
152         ErrorHandler.checkHandle(
153             nameServiceTopic, "DDS.DomainParticipant.create_topic");
154
155         /* Adapt the default SubscriberQos to read from the
156            "ChatRoom" Partition. */
157         status = participant.get_default_subscriber_qos (subQos);
158         ErrorHandler.checkStatus(
159             status, "DDS.DomainParticipant.get_default_subscriber_qos");
160         subQos.value.partition.name = new String[1];
161         subQos.value.partition.name[0] = new String("ChatRoom");
162
163         /* Create a Subscriber for the UserLoad application. */
164         chatSubscriber = participant.create_subscriber(
165             subQos.value, null, ANY_STATUS.value);
166         ErrorHandler.checkHandle(
167             chatSubscriber, "DDS.DomainParticipant.create_subscriber");
168
169         /* Create a DataReader for the NameService Topic
170            (using the appropriate QoS). */
171         parentReader = chatSubscriber.create_datareader(
172             nameServiceTopic,
173             DATAREADER_QOS_USE_TOPIC_QOS.value,
174             null,
175             ANY_STATUS.value);
176         ErrorHandler.checkHandle(
177             parentReader, "DDS.Subscriber.create_datareader (NameService)");
178
179         /* Narrow the abstract parent into its typed representative. */
180         nameServer = NameServiceDataReaderHelper.narrow(parentReader);
181         ErrorHandler.checkHandle(
182             nameServer, "Chat.NameServiceDataReaderHelper.narrow");
183
184         /* Adapt the DataReaderQos for the ChatMessageDataReader to
185            keep track of all messages. */
186         status = chatSubscriber.get_default_datareader_qos(messageQos);
187         ErrorHandler.checkStatus(
```

```
188             status, "DDS.Subscriber.get_default_datareader_qos");
189         status = chatSubscriber.copy_from_topic_qos(
190             messageQos, reliableTopicQos.value);
191         ErrorHandler.checkStatus(
192             status, "DDS.Subscriber.copy_from_topic_qos");
193         messageQos.value.history.kind =
194             HistoryQosPolicyKind.KEEP_ALL_HISTORY_QOS;
195
196         /* Create a DataReader for the ChatMessage Topic
197            (using the appropriate QoS). */
198         parentReader = chatSubscriber.create_datareader(
199             chatMessageTopic,
200             messageQos.value,
201             null,
202             ANY_STATUS.value);
203         ErrorHandler.checkHandle(
204             parentReader, "DDS.Subscriber.create_datareader (ChatMessage)");
205
206         /* Narrow the abstract parent into its typed representative. */
207         loadAdmin = ChatMessageDataReaderHelper.narrow(parentReader);
208         ErrorHandler.checkHandle(
209             loadAdmin, "Chat.ChatMessageDataReaderHelper.narrow");
210
211         /* Initialize the Query Arguments. */
212         params = new String[1];
213         params[0] = new String("0");
214
215         /* Create a QueryCondition that will contain all messages
216            with userID=ownID */
217         singleUser = loadAdmin.create_querycondition(
218             ANY_SAMPLE_STATE.value,
219             ANY_VIEW_STATE.value,
220             ANY_INSTANCE_STATE.value,
221             "userID=%0",
222             params);
223         ErrorHandler.checkHandle(
224             singleUser, "DDS.DataReader.create_querycondition");
225
226         /* Create a ReadCondition that will contain new users only */
227         newUser = nameServer.create_readcondition(
228             NOT_READ_SAMPLE_STATE.value,
229             NEW_VIEW_STATE.value,
230             ALIVE_INSTANCE_STATE.value);
231         ErrorHandler.checkHandle(
232             newUser, "DDS.DataReader.create_readcondition");
233
234         /* Obtain a StatusCondition that triggers only when a Writer
235            changes Liveliness */
236         leftUser = loadAdmin.get_statuscondition();
237         ErrorHandler.checkHandle(
238             leftUser, "DDS.DataReader.get_statuscondition");
239         status = leftUser.set_enabled_statuses(
240             LIVELINESS_CHANGED_STATUS.value);
241         ErrorHandler.checkStatus(
242             status, "DDS.StatusCondition.set_enabled_statuses");
243
244         /* Create a bare guard which will be used to close the room */
245         escape = new GuardCondition();
246
247         /* Create a waitset and add the ReadConditions */
248         userLoadWS = new WaitSet();
```

```
249         status = userLoadWS.attach_condition(newUser);
250         ErrorHandler.checkStatus(
251             status, "DDS.WaitSet.attach_condition (newUser)");
252         status = userLoadWS.attach_condition(leftUser);
253         ErrorHandler.checkStatus(
254             status, "DDS.WaitSet.attach_condition (leftUser)");
255         status = userLoadWS.attach_condition(escape);
256         ErrorHandler.checkStatus(
257             status, "DDS.WaitSet.attach_condition (escape)");
258
259         /* Initialize and pre-allocate the GuardList used to obtain
260            the triggered Conditions. */
261         guardList.value = new Condition[3];
262
263         /* Remove all known Users that are not currently active. */
264         status = nameServer.take(
265             nsList,
266             infoSeq,
267             LENGTH_UNLIMITED.value,
268             ANY_SAMPLE_STATE.value,
269             ANY_VIEW_STATE.value,
270             NOT_ALIVE_INSTANCE_STATE.value);
271         ErrorHandler.checkStatus(
272             status, "Chat.NameServiceDataReader.take");
273         status = nameServer.return_loan(nsList, infoSeq);
274         ErrorHandler.checkStatus(
275             status, "Chat.NameServiceDataReader.return_loan");
276
277         /* Start the sleeper thread. */
278         new UserLoad().start();
279
280         while (!closed) {
281             /* Wait until at least one of the Conditions in the
282                waitset triggers. */
283             status = userLoadWS._wait(guardList, DURATION_INFINITY.value);
284             ErrorHandler.checkStatus(status, "DDS.WaitSet._wait");
285
286             /* Walk over all guards to display information */
287             for (int i = 0; i < guardList.value.length; i++) {
288                 if ( guardList.value[i] == newUser ) {
289                     /* The newUser ReadCondition contains data */
290                     status = nameServer.read_w_condition(
291                         nsList,
292                         infoSeq,
293                         LENGTH_UNLIMITED.value,
294                         newUser);
295                     ErrorHandler.checkStatus(
296                         status,
297                         "Chat.NameServiceDataReader.read_w_condition");
298
299                     for (int j = 0; j < nsList.value.length; j++) {
300                         System.out.println(
301                             "New user: " + nsList.value[j].name);
302                     }
303                     status = nameServer.return_loan(nsList, infoSeq);
304                     ErrorHandler.checkStatus(
305                         status, "Chat.NameServiceDataReader.return_loan");
306
307                 } else if ( guardList.value[i] == leftUser ) {
308                     // Some liveliness has changed (either a DataWriter
309                     // joined or a DataWriter left)
```

```
310                     status = loadAdmin.get_liveliness_changed_status(
311                         livChangStatus);
312                     ErrorHandler.checkStatus(
313                         status,
314                         "DDS.DataReader.get_liveliness_changed_status");
315                     if (livChangStatus.value.alive_count < prevCount) {
316                         /* A user has left the ChatRoom, since a DataWriter
317                            lost its liveliness. Take the effected users
318                            so they will not appear in the list later on. */
319                         status = nameServer.take(
320                             nsList,
321                             infoSeq,
322                             LENGTH_UNLIMITED.value,
323                             ANY_SAMPLE_STATE.value,
324                             ANY_VIEW_STATE.value,
325                             NOT_ALIVE_NO_WRITERS_INSTANCE_STATE.value);
326                         ErrorHandler.checkStatus(
327                             status, "Chat.NameServiceDataReader.take");
328
329                         for (int j = 0; j < nsList.value.length; j++) {
330                             /* re-apply query arguments */
331                             params[0] =
332                                 Integer.toString(nsList.value[j].userID);
333                             status = singleUser.set_query_parameters(params);
334                             ErrorHandler.checkStatus(
335                                 status,
336                                 "DDS.QueryCondition.set_query_parameters");
337
338                             /* Read this users history */
339                             status = loadAdmin.take_w_condition(
340                                 msgList,
341                                 infoSeq2,
342                                 LENGTH_UNLIMITED.value,
343                                 singleUser );
344                             ErrorHandler.checkStatus(
345                                 status,
346                                "Chat.ChatMessageDataReader.take_w_condition");
347
348                             /* Display the user and his history */
349                             System.out.println(
350                                 "Departed user " + nsList.value[j].name +
351                                 " has sent " + msgList.value.length +
352                                 " messages.");
353                             status = loadAdmin.return_loan(msgList, infoSeq2);
354                             ErrorHandler.checkStatus(
355                                 status,
356                                 "Chat.ChatMessageDataReader.return_loan");
357                             msgList.value = null;
358                             infoSeq2.value = null;
359                         }
360                         status = nameServer.return_loan(nsList, infoSeq);
361                         ErrorHandler.checkStatus(
362                             status,
363                             "Chat.NameServiceDataReader.return_loan");
364                         nsList.value = null;
365                         infoSeq.value = null;
366                     }
367                     prevCount = livChangStatus.value.alive_count;
368
369                 } else if ( guardList.value[i] == escape ) {
370                     System.out.println("UserLoad has terminated.");
```

```
371                     closed = true;
372                 }
373                 else
374                 {
375                     assert false : "Unknown Condition";
376                 };
377             } /* for */
378         } /* while (!closed) */
379
380         /* Remove all Conditions from the WaitSet. */
381         status = userLoadWS.detach_condition(escape);
382         ErrorHandler.checkStatus(
383             status, "DDS.WaitSet.detach_condition (escape)");
384         status = userLoadWS.detach_condition(leftUser);
385         ErrorHandler.checkStatus(
386             status, "DDS.WaitSet.detach_condition (leftUser)");
387         status = userLoadWS.detach_condition(newUser);
388         ErrorHandler.checkStatus(
389             status, "DDS.WaitSet.detach_condition (newUser)");
390
391         /* Free all resources */
392         status = participant.delete_contained_entities();
393         ErrorHandler.checkStatus(
394             status, "DDS.DomainParticipant.delete_contained_entities");
395         status = TheParticipantFactory.value.delete_participant(participant);
396         ErrorHandler.checkStatus(
397             status, "DDS.DomainParticipantFactory.delete_participant");
398
399     }
400
401 }
```

Bibliography

Bibliography

The following documents are referred to in the text:

[1] *Data Distribution Service for Real-Time Systems Specification*, Version 1.1, formal/05-12-04, Object Management Group (OMG).

[2] *C Language Mapping Specification*, 99-07-35, June 1999 edtion, OMG.

[3] *OpenSplice DDS C Reference Guide*, Version 2.0, PrismTech Limited.

[4] *OpenSplice DDS Deployment Guide*, Version 2.0, PrismTech Limited.

[5] *OpenSplice DDS IDL Pre-processor Guide*, Version 2.0, PrismTech Limited.

Index

Index

O

P

Q

R

S

T

U