PRISMTECH
AN ADLINK COMPANY

RMI User Guide

Release 6.x

Contents

1 Preface

1.1 About the RMI User Guide

The Vortex OpenSplice *RMI User Guide* is intended to explain the steps required to take advantage of the client/server interaction paradigm provided by Vortex OpenSplice RMI layered over the publish/subscribe paradigm of Vortex OpenSplice.

Intended Audience

This Vortex OpenSplice *RMI User Guide* is for developers using remote invocations in DDS applications.

Organisation

The first two chapters give a general introduction to RMI over DDS.

Building an RMI Application describes the steps involved in building applications using RMI over DDS.

Language mapping for Vortex OpenSplice RMI gives the 'C++ and Java' mapping of the IDL types that can be declared in the RMI services description file.

RMI Interface to DDS topics mapping rules shows how IDL declarations of RMI interfaces are mapped into IDL declarations of the implied DDS topics.

RMI Runtime Configuration Options describes the command-line options available when starting the *RMI* runtime.

QoS policies XML schema contains the XML schema for reference.

1.2 Conventions

The icons shown below are used to help readers to quickly identify information relevant to their specific use of Vortex.

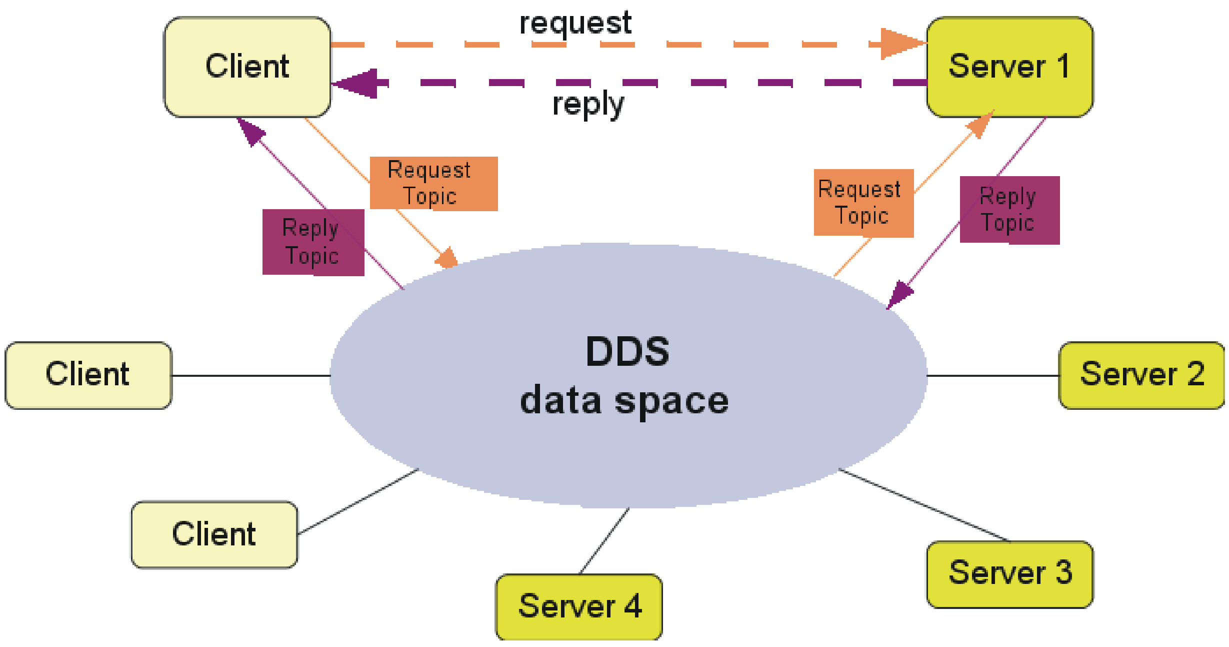
Icon	*Meaning*
⚠	Item of special significance or where caution needs to be taken.
ℹ	Item contains helpful hint or special information.
Windows	Information applies to Windows (*e.g.* XP, 2003, Windows 7) only.
Unix	Information applies to Unix-based systems (*e.g.* Solaris) only.
Linux	Information applies to Linux-based systems (*e.g.* Ubuntu) only.
C	C language specific.
C++	C++ language specific.
C#	C# language specific.
Java	Java language specific.

2

Introduction

2.1 Features

Vortex OpenSplice RMI provides an implementation of the general concept of invoking a remote method over DDS. It enhances Vortex OpenSplice with a service-oriented interaction pattern that can be used with combination with the native data-centric pattern. Vortex OpenSplice RMI is a service invocation framework on top of DDS DCPS that uses DDS mechanisms to export, find and invoke services. It maps all the application-exchanged requests/replies into DDS data exchanges, and gives the ability to configure the associated QoS policies according to the application needs. Finally, Vortex OpenSplice RMI enables the definition of a distributed services space over a DDS data space with all the known DDS benefits, such as discovery, fault tolerance, performance and real-time features.



RMI Communication Scheme

Vortex OpenSplice RMI targets service-oriented applications needing a request/reply communication scheme while they need to have a very fine control over the data and the underlying network quality of service. Typically, Vortex OpenSplice RMI can be used in systems to issue commands. Commands are a kind of stimulus that express the ability of the system to do something. As commands have the ‘do-something’ connotation, it is often useful to be informed synchronously that the command has been executed. Thanks to the various DDS QoSs, applications can associate expiration time, prioritities, persistency and so on to those commands.

2.2 Benefits

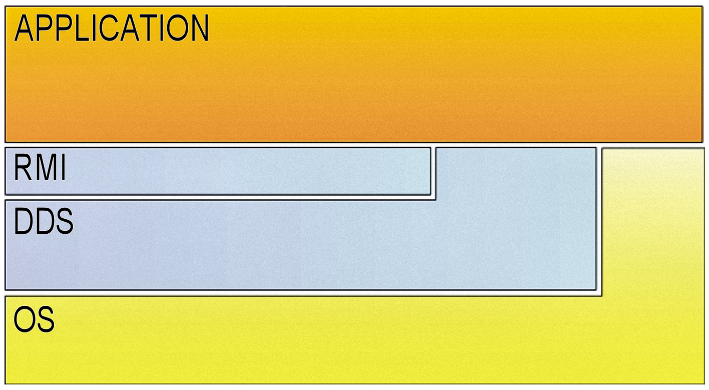
As a complementary paradigm to data centricity, Vortex OpenSplice RMI provides these benefits:

- A more productive and higher abstraction level than can be achieved manually through topic exchanges and applications synchronization.
- A unique middleware technology for mixing Global Services and Data Spaces with an easy and dynamic services registration, data declaration, and the same discovery mechanisms.
- Enables data-centric applications to use RMI without the burden of an additional middleware technology (*e.g.* CORBA).
- Strong services location transparency. Thanks to the connectionless nature of DDS, service identities do not need to include any network-related information. In Vortex OpenSplice RMI, a service is identified by a simple name. Services' identities are exported naturally *via* a DDS publication on specific topics. Services can even move from one location to another without any impact on client applications.
- Simple API.
- Easy deployment process.

3

Vortex OpenSplice RMI over DDS

As in traditional service-oriented applications, communication from client to server is performed through a well-defined service model. The RMI module enables a user to build a service model with remote method invocation capabilities and completely hides the DDS DCPS API. Of course, using RMI does not prevent the application from using the DDS API as shown by the following figure:



RMI Relationship to DDS

A service model is defined by one or more object-oriented interfaces. A DDS RMI interface is an IDL interface having a name and a set of operations. Each operation has a fixed set of typed parameters. The RMI module provides:

- A service invocation framework that maps the different services operations onto a set of DDS topics that hold the operation's invocation requests and replies. A set of mapping rules have been defined for this purpose. At runtime, this framework sets up the underlying DDS environment and handles the remote interface invocations using the basic DDS read/write operations.
- A simple and intuitive programming model for both the server application side implementing the interface, and the client application side invoking that interface. The server programming model is as simple as implementing an interface, and the client programming model is as simple as calling a local interface.
- A powerful feature to enable tuning of the invocation request and reply QoS by setting their corresponding DDS QoS policies. This feature enables developers to improve the invocations quality with real-time and high-performance features. For instance, priorities and validity durations (lifespan) could be set on the different operation requests/replies.
- Synchronous, asynchronous and oneway invocation modes. The synchronous mode is the invocation mode that blocks the client thread until the reply is sent back to him by the server. The asynchronous mode is similar to the CORBA Asynchronous Messaging Interface (AMI) callback model. It is a non-blocking mode where the client does not wait for the reply from the server, but rather provides a callback object that will be invoked by the middleware to deliver the request return values when they are received. Finally, the oneway mode is a fire-and-forget invocation mode where the client does not care about the success or failure of the invocation. A oneway method cannot return values and no reply message will ever return to the client once the request is sent to the server.
- 'C++ and Java' implementation.

3.1 Key components

The Vortex OpenSplice RMI module includes the following components:

RMI Pre-processor (`rmipp`) Generates the interface-type-specific requests/replies topics and invocation handling classes.

Core library Provides the runtime setup operations and a generic invocation framework.

3.2 Binding Languages

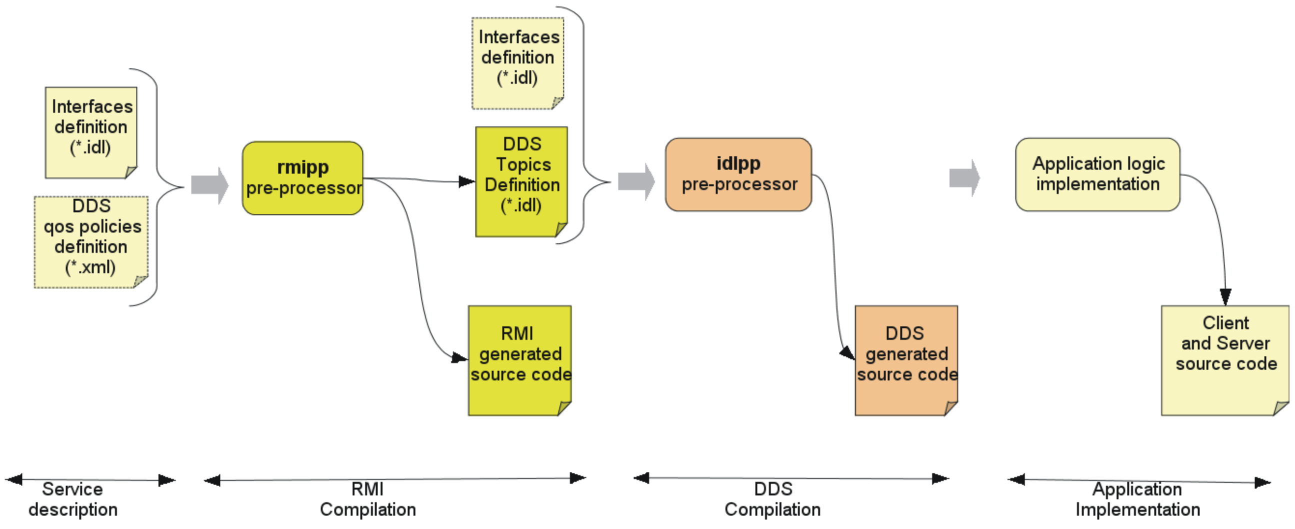
The Vortex OpenSplice RMI module is available for both **Java** and **C++** languages.

4

Building an RMI Application

4.1 About RMI Applications

The process of building an Vortex OpenSplice RMI application is shown in Steps Building Applications with RMI below. The different steps are described in the following subsections.



Interfaces definition (*.idl)
DDS qos policies definition (*.xml)
rmipp pre-processor
Interfaces definition (*.idl)
DDS Topics Definition (*.idl)
RMI generated source code
idlpp pre-processor
DDS generated source code
Application logic implementation
Client and Server source code
Service description
RMI Compilation
DDS Compilation
Application Implementation

Steps Building Applications with RMI

4.2 Services description

The first step in building an RMI application is the definition of its provided services in terms of interfaces. The application interfaces should be declared using the OMG IDL language. The operations parameters can be either of basic (`short`, `long`, ...) or complex (`struct`, `sequence`, `string`, ...) types. However, the following restrictions should be respected:

- the `Any` and the `valuetype` IDL types are not supported because they are not supported by the underlying DDS DCPS layer. `Union` type is also not supported.
- Exceptions are not supported at this time.
- Each interface must extend `DDS_RMI::Services` base interface to indicate that it is invocable over DDS. This interface is defined in the file `dds_rmi.idl`, which must be included.
- Each interface must be declared 'local'.
- Oneway operations are supported. The semantics of oneway operations is the same as for the OMG CORBA interfaces. A oneway operation must not contain any output parameter and must return a void type.

The following IDL snippet shows an example of a service data description:

```
#include "dds_rmi.idl"
module HelloWorld
{
  // interface definition
local interface HelloService : ::DDS_RMI::Services
{
  string greet();
};
};
```

4.3 QoS policies description

The Vortex OpenSplice RMI module provides the ability to tune the quality of service of the services invocations (requests and/or replies), if needed, by setting the underlying DDS QoS policies. By default, the DDS RMI module uses the default values of the DDS QoS policies except for the reliability QoS policy which is set to `RELIABLE`.

If needed, the application designer can define the QoS policies to be set on the invocations in an XML file. This file must respect the XML schema given in *QoS policies XML schema*.

Note that setting the DDS QoS policies requires a good knowledge of the rules for mapping the specified interfaces onto the DDS topics description (please refer to *RMI Interface to DDS topics mapping rules*).

The following XML snippet shows an example:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<dcps xmlns="http://www.omg.org/dds/"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.omg.org/dds/DCPS.xsd">

<domain id="">
  <topic name="greet_req"
idltype="::HelloWorld::HelloService::greet_request" idlfile="">

  <topic_qos>
  <destinationOrderQosPolicy>
    <destinationOrderKind>
      BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS
    </destinationOrderKind>
  </destinationOrderQosPolicy>

  <durabilityQosPolicy>
    <durabilityKind>
      PERSISTENT_DURABILITY_QOS
    </durabilityKind>
  </durabilityQosPolicy>

  <latencyBudgetQosPolicy>
    <duration>
      <nanosec>10000000</nanosec>
      <sec>0</sec>
    </duration>
  </latencyBudgetQosPolicy>

  <reliabilityQosPolicy>
    <duration>
      <nanosec>100000000</nanosec>
      <sec>0</sec>
    </duration>
  <reliabilityKind>
    RELIABLE_RELIABILITY_QOS
  </reliabilityKind>
  </reliabilityQosPolicy>

  </topic_qos>

  </topic>

</domain>
</dcps>
```

This example specifies the QoS policies to be applied on the topic invocation request of the `greet` operation of the interface `HelloWorld::HelloService`. Note that the invocation request topic is named `greet_req` and its IDL type is `HelloWorld::HelloService::greet_request`.

4.4 RMI compilation

Once the application has defined its services and (optionally) its QoS settings, these definitions are compiled to generate type-specific code for the application services invocation.

The RMI compilation is done using the `rmipp` pre-processor applied on the interfaces definition file and the QoS file if it exists. The `rmipp` usage is:

```
rmipp [-l (java | c++)] [-I <path>] [-d <directory>] [-topics
<qos_file>] [-P dll_macro_name[,<header-file>]] <interfaces_file>
```

The parameters are:

-l (java | c++) Define the target language. The C++ language is the default.

-I <path> Define the include path directives.

-d <directory> Define the location to place the generated files.

-topics <qos_file> Define the XML file including the QoS policies settings.

-P dll_macro_name[,<header-file>] *Only applicable to C and C++.* Sets export macro that will be prefixed to all functions in the generated code. This allows creating DLLs from generated code. Optionally a header file can be given that will be included in each generated file.

<interfaces_file> The IDL file including the interfaces definition.

The `rmipp` compilation will generate a set of Java or C++ source files as well as an IDL file including the mapping of the provided interfaces onto the DDS topics. The generated IDL file name is the interfaces file name with “`_topics`” concatenated.

`Rmipp` follows the mapping rules described in *Language mapping for Vortex OpenSplice RMI*.

Example usage:

```
rmipp -d generated HelloService.idl
```

The generated directory will include:

```
HelloService_topics.idl
HelloService_Interface.h
HelloService_Interface.cpp
HelloService_InterfaceProxy.h
HelloService_InterfaceProxy.cpp
```

In addition, the `rmipp` compiler performs a DDS compilation to generate the DDS/DCPS code that is required to support the requests/replies transport over DDS.

4.5 Application implementation

As mentioned before, the target applications have a client/server design. A typical application includes a server part that implements the provided interfaces, and a client part that invokes these interfaces. This section describes the programming model of both parts.

4.5.1 Runtime starting and stopping

Any DDS RMI application process must initialize the RMI runtime prior to any other operation, regardless of whether it is a client and/or a server process. The runtime initialization sets up the underlying DDS infrastructure and configures it to make the services invocable and the clients capable of invoking the services. It is also important to stop the runtime when the application is no longer using RMI.

The following code snippets show the runtime initialisation and stopping procedure in C++ and Java.

RMI runtime starting and stopping in C++

```
01  #include "ddsrmi.hpp"
02
03  using namespace org::opensplice::DDS_RMI;
04
05  int main (int argc, char * argv [])
06  {
07    CRuntime_ref runtime = CRuntime::getDefaultRuntime();
08    if (runtime.get() == NULL)
09    {
10      std::cout << "Failed to get the Runtime " << std::endl;
11      exit(1);
12    }
13
14    //starting the runtime
15    bool result = runtime->start(argc, argv);
16    if (result !=true)
17    {
18      std::cout << "Failed to start the Runtime " << std::endl;
19      exit(1);
20    }
21    ...
22
23    //stopping the runtime
24    result = runtime->stop();
25    if (result !=true)
26    {
27      std::cout << "Failed to stop the Runtime " << std::endl;
28      exit(1);
29    }
30  }
```

Comments below refer to line numbers in the sample code above:

1 ***Include the OpenSplice RMI library header file.*** Any OpenSplice RMI application should include this file.

3 *Declare the usage of the OpenSplice RMI library namespace.*

7-12 ***Get the default DDS runtime.*** This selects the default DDS domain as the data space where all subsequent RMI requests and replies will be exchanged.

15-20 ***Initialize the created runtime.*** This creates all the needed DDS entities. A set of configuration options can be passed to the `start` operation *via* `argc` and `argv` parameters. This latter is a string array including possible option names and values, and `argc` is the length of this array. Note that these parameters are typically the same parameters that were passed to the main program so that the RMI options can be specified on the command line, each following the format '`--option=value`'. All of the supported options are described in the section *RMI Runtime Configuration Options*.

24-28 ***Stop the created runtime.*** This removes all the created DDS entities and releases the RMI-allocated resources. It is strongly recommended to stop the runtime when it no longer needed.

The Java code below works in a similar way.

Java

RMI runtime starting and stopping in Java

```
import org.opensplice.DDS_RMI;

static void main (String[] args)
{

  CRuntime runtime = CRuntime.getDefaultRuntime();
  if(null == runtime)
  {
    System.out.println();
    System.exit(1);
  }

  //starting the runtime
  boolean result = runtime.start(args);
  if(!result)
  {
    System.out.println("Failed to start the Runtime") ;
    System.exit(1);
  }
  ...
  //stopping the runtime
  result = runtime.stop();
  if(!result)
  {
    System.out.println("Failed to stop the Runtime") ;
    System.exit(1);
  }
}
```

4.5.2 Server programming model

At the server side of the application, each provided interface should be implemented, then instantiated and finally registered to be invocable *via* Vortex OpenSplice.

To define an implementation, the application developer must write an implementation class including public methods corresponding to the operations of the related IDL interface. The `rmipp` compilation generates for each interface a skeleton class, named `::DDS_RMI::HelloWorld::HelloServiceInterface`, that must be extended by the application-supplied implementation class. The language mapping rules of the RMI IDL interfaces are given in *Language mapping for Vortex OpenSplice RMI*.

To make an interface invocable over DDS, it must be registered within the RMI framework, then activated. The registration process requires the following information:

- the implementation class object
- the server name, as well as a unique id identifying that interface inside the server.

The services activation makes the RMI runtime wait for incoming requests for all the registered services.

The following code snippets show the server programming model in C++ and Java.

C++

C++ RMI interface implementation

```
class HelloService_impl :
      public virtual DDS_RMI::HelloWorld::HelloServiceInterface
{
public:
    HelloService_impl();
   ~ HelloService_impl();

   virtual DDS::String greet ();
}
```

Java

Java RMI interface implementation

```
public class HelloService_impl :
DDS_RMI.HelloWorld.myInterfaceInterface {
   public String greet ()
   {
      // operation implementation
   }
}
```

C++

C++ RMI server

```
#include "ddsrmi.hpp"
#include "HelloService_Interface.hpp"

using namespace org::opensplice::DDS_RMI;

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

   // Runtime starting
   ...

   // implementation class instantiation
   shared_ptr<HelloService_impl> impl (new HelloService_impl());

   //interface registration
   bool res = DDS_Service::register_interface<
   ::DDS_RMI::HelloWorld::HelloServiceInterface, HelloService_impl>
   (
         impl, //implementation class
         "HelloServer", // server name
         1 // unique server id
    );

   if(!res)
   {
       std::cout << "Failed to register the
                     HelloWorld::HelloService interface") ;
       System.exit(1);
   }
   //services activation
   runtime->run()
   // Runtime stopping
   ...
```

```
32   }
```

Comments below refer to line numbers in the sample code above:

1-2 *Include the OpenSplice RMI library header file as well as the generated interface skeleton header file.*

4 *Declare the usage of the OpenSplice RMI library namespace.*

10 *Start the DDS runtime.*

13 ***Instantiate the implementation class of the*** **`HelloService`** ***interface and assign it to a smart pointer.***
The OpenSplice RMI library provides an implementation of smart pointers *via* the `shared_ptr` template class.

16-27 ***Register the*** **`HelloService`** ***interface in the default DDS domain.*** The `register_interface` function is a template function requiring the interface skeleton class and the interface implementation class as template parameters.

28 ***Activates all the registered services including the*** **`HelloServer`** ***service.*** This is a blocking call that makes the server runtime wait for incoming requests. To shut down the server runtime the `shutdown()` operation must be called.

31 *Stop the DDS runtime.*

The Java code below works in a similar way.

Java

Java RMI server

```
static void main (String[] args)
{

   // Runtime starting
   ...

   // implementation class instanciation
   HelloService_impl impl = new HelloService_impl();

   // interface registration
   boolean res = org.opensplice.DDS_RMI.DDS_Service.register_interface
   (
         impl, // implementation class
         "HelloServer", // server name
         1, // unique server id
         DDS_RMI.HelloWorld.HelloServiceInterface.class //Interface java Class
   );

   if(!res)
   {
       System.out.println("Failed to register the
                           HelloWorld::HelloService interface") ;
       System.exit(1);
   }
runtime.run();
   // Runtime stopping
   ...
}
```

4.5.3 Client programming model

As mentioned before, OpenSplice RMI supports synchronous, asynchronous and oneway invocation modes. The following subsections present the synchronous and asynchronous programming model. The oneway programming model is similar to the synchronous one but, of course, with a different behaviour.

Synchronous invocation mode

The client part of the RMI application is as simple as calling a local class. Note that these calls block until the server-side responds or an internal timeout expires. Typically, in case of failure, the call will block until the timeout expiration. This timeout value is set by default to 10 minutes, but it may be configured *via* the interface proxy object. This object is a generated object, named `::DDS_RMI::HelloWorld::HelloServiceInterfaceProxy`, that is the local representative of the RMI interface. This object is mainly used to call the RMI services, as shown in the following client code examples.

C++ RMI client

```
01 #include “ddsrmi.hpp”
02 #include “HelloService_InterfaceProxy.hpp”
03
04 using namespace org::opensplice::DDS_RMI;
05
06 int main (int argc, char * argv [])
07 {
08
09   // Runtime starting
10   ...
11
12   // Getting the interface proxy
13   shared_ptr<::DDS_RMI::HelloWorld::HelloServiceInterfaceProxy> proxy ;
14   bool ret = DDS_Service::getServerProxy<
14               ::DDS_RMI::HelloWorld::HelloServiceInterfaceProxy>
15     (
16       “HelloServer”, //server name
17       1, //unique proxy instance id
18       proxy  // proxy reference
19     );
20
21   // Calling the services
22   proxy->greet();
23
24   // Runtime stopping
25   ...
26
27 }
```

Comments below refer to line numbers in the sample code above:

1-2 *Include the RMI library header file as well as the generated interface proxy header file.*

4 *Declare the usage of the OpenSplice RMI library namespace.*

10 *Start the DDS runtime.*

13 *Declare a smart pointer of the HelloService interface proxy type.*

13-19 ***Get the HelloServer service proxy.*** The `getServerProxy` function is a template function requiring the proxy class type as a template parameter. This function accepts the service name, a proxy instance id and the smart pointer to the proxy object as parameters. In case of success, the smart pointer is set to the created proxy object. The proxy instance id is a unique identifier that refers to the created proxy. It is important to ensure the uniqueness of the identifiers of all the proxies of the same service. If the client application intends to use the same proxy in different threads, the MultiThreaded mode must be set (see MultiThreaded Client later in this chapter). If the requested service is not found, the `getServerProxy` operation will raise an `org::opensplice::DDS_RMI::SERVICE_NOT_FOUND` exception.

22 *Invoke the greet operation synchronously using the created proxy.*

25 *Stop the runtime.*

The Java code below works in a similar way.

Java

Java RMI client

```
import org.opensplice.DDS_RMI.*;

static void main (String[] args) {

  // Runtime starting
  ...

  // Getting the interface proxy
  try {
  DDS_RMI.HelloWorld.HelloServiceInterfaceProxy proxy =
    DDS_Service.getServerProxy (
      "HelloServer", //server name
      1, //unique proxy instance id
       DDS_RMI.HelloWorld.HelloServiceInterfaceProxy.class // proxy java Class
    );

  // Calling the services
  proxy.greet();
  } catch (SERVICE_NOT_FOUND e) {
  // error
  }
  // Runtime stopping
  ...

}
```

Asynchronous invocation mode

To invoke asynchronously a given non-oneway operation, such as the `greet` operation in the examples shown here, the client application must:

- Implement a specific reply handler class to handle the operation out/inout/return parameters if any. This handler must extend a base reply handler class that is generated for each operation and implement the `greetReply` callback function or method whose parameters are the out/inout/return parameters of the related IDL operation.
- Use the generated asynchronous function or method that maps to the IDL operation whose name is the concatenation of '`async_`' and the IDL operation name. This operation is a void operation that accepts only the `in` and `inout` IDL parameters, in addition to the eference of the implemented reply handler.

Note that the reply handler class is not re-entrant in the current implementation. It cannot handle concurrent replies. It means that if two successive asynchronous calls are made with the same reply handler instance, this latter will reject the second reply if it has not finished dispatching the first one. In this case the asynchronous call will raise a `BAD_PARAM` exception.

IMPORTANT: It is strongly recommended not to mix synchronous and asynchronous calls of the same operation without proper synchronization. The application should ensure that the asynchronous call has received its reply before requesting a synchronous one.

C++

C++ RMI Client with asynchronous invocation

```
#include "ddsrmi.hpp"
#include "HelloService_InterfaceProxy.hpp"

```

```
04   using namespace org::opensplice::DDS_RMI;
05
06   /**
07   * Reply Handler of the 'async_greet' operation
08   *
09   */
10   class MyGreetReplyHandler :
11      public virtual   HelloWorld_HelloService_greet_Reply_Handler
12   {
13      void greet_Reply(DDS::String ret)
14      {
15           std::cout << "Reply received: " << ret << std::endl;
16      }
17   }
18
19   int main (int argc, char * argv [])
20   {
21
22   // Runtime starting
23      ...
24
25   // Getting the interface proxy
26   shared_ptr<::DDS_RMI::HelloWorld::HelloServiceInterfaceProxy> proxy ;
27   bool ret = DDS_Service::getServerProxy<
27              ::DDS_RMI::HelloWorld::HelloServiceInterfaceProxy>
28      (
29        "HelloServer", //server name
30        1, // proxy instance id
31        proxy  // proxy reference
32      );
33
34   // instantiating a reply handler
35   MyGreetReplyHandler handler;
36
37   // Calling the services asynchronously
38   proxy->async_greet(&handler);
39   ...
40
41   // Runtime stopping
42   ...
43
44   }
```

Comments below refer to line numbers in the sample code above:

10-16 *Provide the implementation class of the* `greet` *operation reply handler.*

21 *Start the DDS runtime.*

24-31 *Get the* `HelloServer` *service proxy as for the synchronous mode.*

34 *Instantiate the* `greet` *reply handler class.*

37 ***Invoke the*** **`async_greet()`** ***operation by providing the reply handler.*** This call is a non-blocking call. The application steps immediately to the next instruction. The invocation reply will be delivered to the application by invoking the `greet_Reply` operation of the reply handler. Note that this operation will be invoked in a middleware-provided thread.

41 ***Stop the runtime.*** Note that some synchronization may be needed to avoid exiting before the `async_greet` reply is delivered to the application.

The Java code below works in a similar way.

Java

Java RMI Client with asynchronous invocation

```
import org.opensplice.DDS_RMI.*;

/**
     * Reply Handler of the 'async_greet' operation
     *
     */
    class MyGreetReplyHandler extends
          DDS_RMI.HelloWorld.HelloServiceInterfaceProxy.greet_Reply_Handler {
        public void greet_Reply(String ret) {
            System.out.println("async_greet returns: " + ret);
        }
    };

static void main (String[] args) {

   // Runtime starting
   ...

  try {
   // Getting the interface proxy
   DDS_RMI.HelloWorld.HelloServiceInterfaceProxy proxy =
      DDS_Service.getServerProxy (
        "HelloServer", //server name
        1, //server instance id
         DDS_RMI.HelloWorld.HelloServiceInterfaceProxy.class // proxy java Class
      );

   // Calling the services asynchronously
   proxy.asynch_greet();
  } catch(SERVICE_NOT_FOUND e) {
    System.out.println("'HelloServer' service not found !");
  }

   // Runtime stopping
   ...

}
```

MultiThreaded Client

The default threading model of a client application is single threaded. It means that, by default, a service proxy may not be used by multiple concurrent threads to perform service invocations. To enable or disable the multithreaded mode for clients, a configuration option must be specified in the command line as follows:

```
--RMIClientThreadingModel=[ST|MT]
```

4.6 Using a specific DDS domain

Using the default RMI CRuntime implies that all the RMI invocations will be performed within the default DDS domain. The default domain id is the one specified by the current Vortex OpenSplice configuration, specifically in the Domain Service section of the related XML file. The default Vortex OpenSplice configuration files set the `domain id` to `0`. For more information on configuring Vortex OpenSplice please refer to the *Vortex OpenSplice Deployment Guide*.

If the RMI application operates with a user-defined domain id, using the default CRuntime enables RMI interactions within that domain. In case of an application operating in multiple domains, it should create a CRuntime object for the targeted domain id and get a `DDS_ServiceInterface` object from that CRuntime.

The `DDS_ServiceInterface` object provides all the convenient methods for server applications to register/unregister services and for client applications to get service proxies on the relevant domain id. These methods are the same as the `DDS_Service` object ones. The following code snippets show this in C++ and Java.

C++

Getting DDS_ServiceInterface in C++

```
// Getting a CRuntime on my specific domain
CRuntime_ref runtime = CRuntime::getRuntime(my_domain_id);
// Getting a DDS_ServiceInterface object
DDS_ServiceInterface_ref dds_service = runtime->getDDS_ServiceInterface();
```

Java

Getting DDS_ServiceInterface in Java

```
// Getting a CRuntime on my specific domain
CRuntime runtime = CRuntime.getRuntime(my_domain_id);
// Getting a DDS_ServiceInterface object
DDS_ServiceInterface dds_service = runtime.getDDS_ServiceInterface();
```

4.7 Server Threading and Scheduling policies

OpenSplice RMI allows configuration of the threading and the scheduling models of the RMI server applications by enabling a set of policies that control how the server allocates threads to handle service invocations and how these threads are scheduled with regard to the others. Hence developers may enhance the responsiveness of their services by choosing a multi-threaded execution model, or may protect a non-thread-safe service implementation by choosing a single thread execution model.

4.7.1 Threading policies

OpenSplice RMI provides three threading policies that apply on a single RMI runtime at the server side. The RMI runtime uses a thread pool that hosts a number of threads to execute the services incoming requests. This number defines the thread pool size and depends on the specified threading policy.

1 – Single Thread (ST) policy This policy allocates a single thread to process all the services' incoming requests within the server process. The requests are processed one by one in the order that they are received. It guarantees that no two threads will execute concurrently the different services calls within that server. In this case, the RMI runtime creates a single thread pool. This policy is the default threading policy. The ST policy ensures thread safety to all the services within the server process, but with the drawback that services with long-running tasks will block the others.

2 – Thread Per Service (TPS) policy This policy allocates a single thread to each service, up to the number specified as the thread pool size. All the requests coming to one service are processed in sequence by a single thread borrowed from the thread pool, but different services requests are processed in parallel, within the limit of the thread pool size. The thread pool size is defined by the user application. The TPS policy ensures that services having long-running requests won't block the others, but concurrent calls are not possible for one single service.

3 – Multi Thread (MT) policy This policy allows for a full multi-threaded environment. All the incoming requests are processed concurrently in different threads whatever the targeted services, within the limit of the thread pool size. The service implementation should take care of that and ensure thread safety. The thread pool size is user-defined.

Scheduling policies

These policies specify the scheduling parameters that will be used for the threads created by the RMI runtime for a RMI server.

A RMI scheduling policy is defined by a scheduling priority and a scheduling class. The scheduling priority specifies the priority that will be assigned to all the threads that are spawned by the RMI runtime. The scheduling class may be `SCHEDULE_DEFAULT`, `SCHEDULE_TIMESHARING`, or `SCHEDULE_REALTIME`. These scheduling classes depend on the underlying operating system. A `SCHEDULE_DEFAULT` class is the default OS scheduling algorithm.

In general, the Timesharing class attempts to distribute the processor resources fairly among the threads. In a Realtime class a thread normally runs until completion but can be pre-empted by higher-priority threads. Both Timesharing and Realtime scheduling classes are priority-based, so the scheduling priority is meaningful for both classes.

Note that some scheduling classes may not be supported by the underlying operating system, or that you may need special privileges to select particular settings.

Java Note that the Java platform does not define a scheduling model for threads but defers to the underlying thread implementation. As a result, only the scheduling priority is usable for the Java RMI applications. In fact, there *is* a notion of priority in the Java threading model, but this is only a hint to the scheduler. The way in which the JVM maps these hints to the underlying OS scheduler varies from JVM to JVM and even from platform to platform for a given JVM.

Service priority

OpenSplice RMI allows priorities to be assigned to the services registered in a RMI runtime within a server process. It defines the business importance of each service relative to the others. It allows incoming requests to be handled in order of priority if not enough threads are available to handle them concurrently. The service priority is exploited by the OpenSplice RMI framework to decide which service request a thread should be assigned to first, whereas the scheduling priority is exploited by the OS scheduler itself to decide which thread should run first.

By default, each RMI service has priority set to `0`.

Programming model

Threading and scheduling policies may be passed either by command line, or programmatically at runtime. The command line options are described in *RMI Runtime Configuration Options*; this section shows the related APIs only.

Java

Setting the threading/sceduling policies in Java

```java
// getting and starting the default runtime
CRuntime runtime = Cruntime.getDefaultRuntime();
boolean res = runtime.start(argv);

//setting a MT threading policy with a thread pool size set to 5
ServerThreadingPolicy t_policy = new  ServerThreadingPolicy
                                 (ThreadingPolicyKind.MT, 5);
runtime.setServerThreadingPolicy(t_policy);

// setting a scheduling priority and keeping the scheduling
// class to the default
SchedulingPolicy s_policy = runtime.getServerSchedulingPolicy();
s_policy.schedulingPriority = 10;
runtime.setServerSchedulingPolicy(s_policy);

```

```java
//registering a set of services
HelloService_impl impl = new HelloService_impl();
res   = DDS_Service.register_interface(impl, "HelloServer",1,
       DDS_RMI.HelloWorld.HelloServiceInterface.class);
// registering other services

// making HelloServer service the highest priority service
impl.setPriority(2);

//running the runtime
runtime.run();
```

Note that setting the *thread pool size* on java is done asynchronously. This means that there could be more server threads (and thus parallel calls) then expected when reducing the *thread pool size*.

C++

Setting the threading/sceduling policies in C++

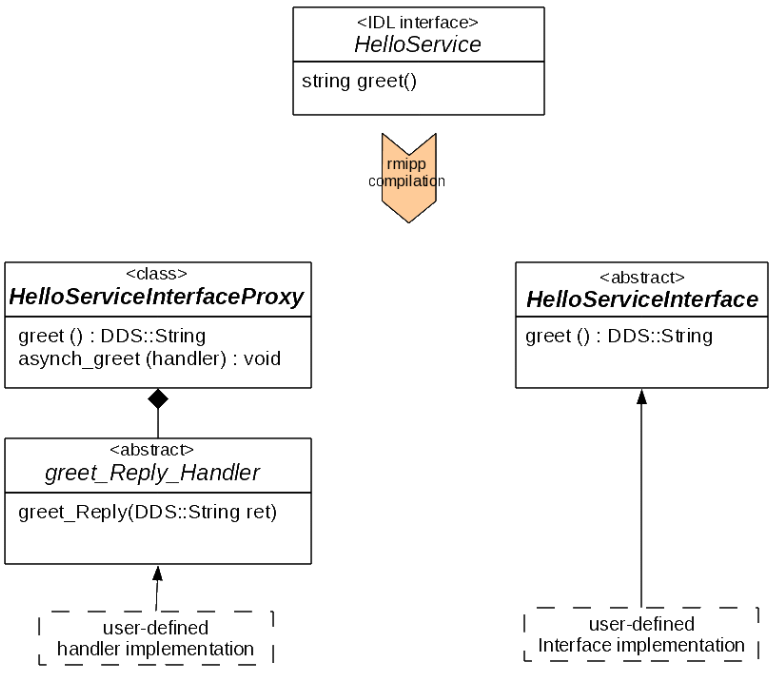
```cpp
// getting and starting the default runtime
CRuntime_ref runtime = Cruntime::getDefaultRuntime();
bool res = runtime.start(argc, argv);

//setting a MT threading policy with a thread pool size set to 5
ServerThreadingPolicy t_policy = runtime->getServerThreadingPolicy();
t_policy.kind = MT;
t_policy.threadPoolSize = 5;
runtime->setServerThreadingPolicy(t_policy);

// setting a scheduling priority and keeping the scheduling
// class to the default
SchedulingPolicy s_policy = runtime->getServerSchedulingPolicy();
s_policy.schedulingPriority = 10;
runtime.setServerSchedulingPolicy(s_policy);

//registering a set of services
shared_ptr<HelloService_impl> impl (new HelloService_impl());
res   = DDS_Service::register_interface<DDS_RMI::HelloWorld::HelloServiceInterface,
       HelloService_impl> (impl, "HelloServer",1);
// registering other services

// making HelloServer service the highest priority service
impl->set_priority(2);

//running the runtime
runtime->run();
```

Note that the *threading policy* may only be changed as long as the runtime is not run. Once run, only the *thread pool size* may be changed.

5 Language mapping for Vortex OpenSplice RMI

Rmipp compilation follows a set of mapping rules to generate language-specific source code. Most of these rules come from the standard OMG IDL-to-C++ and IDL-to-Java mapping specifications but with some specific differences. This chapter focuses on specific parts of this mapping. For more information, please refer to the relevant OMG specifications.

The following figure shows the language mapping of the `HelloService` IDL interface previously defined.



IDL Interface Mapping

5.1 Mapping for interfaces

An interface is mapped to two C++ (or Java) classes that contain public definitions of the operations defined in the interface.

The `HelloServiceInterface` abstract class is the base class of the `HelloService` implementation class. The `HelloServiceInterfaceProxy` class is the proxy object that represents locally the remote service. The client application should get a reference to this class to be able to invoke the remote service.

5.2 Mapping for operations

Each IDL operation, if not oneway, is mapped to two C++ functions (Java methods). The first one, having the same name as the IDL operation, is used for synchronous invocations. The second one, having `async_` concatenated to the IDL operation, is used for asynchronous invocations. A oneway operation maps only to the synchronous form of the operations.

The operations parameters and return types obey the same parameter passing rules as for the standard OMG IDL-to-C++ and IDL-to-Java mapping. The asynchronous functions (methods) will return void and take only the in/inout parameters of the IDL operation, as well as a callback object used as a reply handler. This handler class is also generated for each non-void operation as an inner abstract class of the proxy class as depicted in the diagram with the `greet_Reply_Handler` class. This latter should be implemented by the user to handle the asynchronous invocation reply. Hence, the `greet_Reply` function (method) provides all the inout/out/return parameters of the corresponding IDL operation.

5.3 Mapping for basic types

The table below shows the 'C++ and Java' mapping of the IDL types that can be declared in the RMI services description file.

IDL sequences are mapped as specified by the DDS standard.

Mapping for basic types

IDL type	C++	Java
boolean	DDS::Boolean	boolean
char	DDS::Char	char
octet	DDS::Octet	byte
short	DDS::Short	short
unsigned short	DDS::UShort	short
long	DDS::Long	int
unsigned long	DDS::ULong	int
long long	DDS::LongLong	long
unsigned long long	DDS::ULongLong	long
float	DDS::Float	float
double	DDS::Double	double
string	DDS::String	String

6

RMI Interface to DDS topics mapping rules

This chapter demonstrates the mapping rules driving the transformation of the IDL declarations of the RMI interfaces into the IDL declarations of the implied DDS topics.

- For each `<InterfaceName>`, a new module is created with the same name and scope in the module `DDS_RMI`, where all the topics associated with the interface operations will be made.
- Each `<InterfaceName>.<operation name>` creates two data structures, suffixed respectively with `_request` for the data structure that handles the request, and `_reply` for the data structure that handles the reply.
- The `<operation name>_request` data struct will gather all `[in]` or `[inout]` parameters.
- The `<operation name>_reply` data struct will gather the return value and all `[inout]` or `[out]` parameters.
- `req_info` is used to enable the client service handler to pick the reply it is waiting for.

```
module HelloWorld {
   local interface HelloService : ::DDS_RMI::Services
   {
      void op1 (in string p1, inout short p2, out long p3);
   };
};
```

```
module DDS_RMI {
  module HelloWorld {
    module HelloService {

   struct op1_request {
           DDS_RMI::Request_Header req_info;
           string p1;
           short p2;
        };
   #pragma keylist op1_request req_info.client_id.client_impl
                   req_info.client_id.client_instance

       struct op1_reply {
           DDS_RMI::Request_Header req_info;
           short p2;
           long p3;
       };
   #pragma keylist op1_reply req_info.client_id.client_impl
                   req_info.client_id.client_instance
        };
    };
};
```

7

RMI Runtime Configuration Options

The RMI runtime can be configured by a set of command line options. These options are passed directly to the runtime start operation as described in the section *Runtime starting and stopping*.

This chapter describes the set of supported options.

7.1 RMIClientThreadingModel option

```
--RMIClientThreadingModel = [ST | MT]
```

This option specifies the threading model of a given client. The `ST` and `MT` option values set respectively the Single-Threaded and Multi-Threaded models.

7.2 RMIServiceDiscoveryTimeout option

```
--RMIServiceDiscoveryTimeout = <seconds>
```

This is a client-side option that specifies the maximum duration (in seconds) that a client application can wait to find services. It influences the execution time of the DDS_Service.getServerProxy operation that is used to find a given service. The default value is set to 10 seconds. The need to set this value may come from some specific deployment environements with bad communication conditions.

7.3 RMIServerThreadingModel option

```
--RMIServerThreadingModel=ST | MT | TPS [,<thread-pool-size>]
```

This is a server-side option that specifies the threading policy of the server runtime including the threading policy name and the thread pool size.

ST selects *Single Threaded* policy.

MT selects *Multi Thread* policy.

TPS selects *Thread Per Service* policy.

These policies are described in detail in the section *Server Threading and Scheduling policies*.

7.4 RMIServerSchedulingModel option

```
--RMIServerSchedulingModel=<priority>
```

This is a server-side option that specifies the scheduling policy of a Java server RMI runtime.

7.5 RMIDurability option

Note: The `RMIDurability` option is currently *only* implemented for *C++*.

C++

```
--RMIDurability = yes | no
```

This is a client-side and server-side option that indicates whether the underlying DDS middleware support the non-default durability Qos policies (TRANSIENT_LOCAL and above) or not.

By default, this option value is `yes`.

RMI servers uses non-volatile topics for services advertising to allow late-joining clients to discover them. This option is useful for adapting services registration and discovery mechanisms when the durability support is missing in the underlying DDS middleware.

Note that this feature must be *either* enabled *or* disabled for *all* of the RMI applications in a given DDS domain. It means that durability-*en*abled (option value is `yes`) RMI applications cannot be deployed with durability-*dis*abled (option value is `no`) RMI applications in the same DDS domain.

7.6 RMIClientSchedulingModel option

Note: The `RMIClientSchedulingModel` option is currently *only* implemented for *Java*.

Java

```
--RMIClientSchedulingModel=<priority>
```

This is a client-side option that specifies the priority of all the threads created by OpenSplice RMI at the client side, including the AsyncWaiter thread, which is the one that waits for asynchronous replies.

8

QoS policies XML schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.omg.org/dds/"
            xmlns="http://www.omg.org/dds/"
            elementFormDefault="qualified">
    <xsd:element name="dcps">
        <xsd:complexType>
            <xsd:all>
                <xsd:element ref="domain" minOccurs="1" maxOccurs="1"/>
            </xsd:all>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="domain">
        <xsd:complexType>
            <xsd:sequence>
                <xsd:element ref="topic" minOccurs="1" maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="id" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="topic">
        <xsd:complexType>
            <xsd:all>
                <xsd:element ref="keylist" minOccurs="1" maxOccurs="1"/>
                <xsd:element ref="topic_qos" minOccurs="0" maxOccurs="1"/>
            </xsd:all>
            <xsd:attribute name="name" type="xsd:string" use="required"/>
            <xsd:attribute name="idltype" type="xsd:string" use="required"/>
            <xsd:attribute name="idlfile" type="xsd:string" use="required"/>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="keylist">
        <xsd:complexType>
          <xsd:sequence>
            <xsd:element ref="keyMember" minOccurs="0" maxOccurs="unbounded"/>
          </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="keyMember" type="xsd:string"/>

     <xsd:element name="topic_qos">
       <xsd:complexType>
        <xsd:all>
            <xsd:element ref="topicDataQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="deadlineQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="durabilityQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="durabilityServiceQosPolicy" minOccurs="0"
                             maxOccurs="1"/>
            <xsd:element ref="latencyBudgetQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="livelinessQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="reliabilityQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="destinationOrderQosPolicy" minOccurs="0"
                             maxOccurs="1"/>
            <xsd:element ref="historyQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="resourceLimitsQosPolicy" minOccurs="0"
                             maxOccurs="1"/>
            <xsd:element ref="transportPriorityQosPolicy" minOccurs="0"
                             maxOccurs="1"/>
            <xsd:element ref="lifespanQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="ownershipQosPolicy" minOccurs="0" maxOccurs="1"/>
            <xsd:element ref="timeBasedFilterQosPolicy" minOccurs="0"
                             maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="deadlineQosPolicy">
       <xsd:complexType>
          <xsd:all>
            <xsd:element ref="duration" minOccurs="1" maxOccurs="1"/>
          </xsd:all>
       </xsd:complexType>
    </xsd:element>

    <xsd:element name="timeBasedFilterQosPolicy">
       <xsd:complexType>
          <xsd:all>
            <xsd:element ref="duration" minOccurs="1" maxOccurs="1"/>
          </xsd:all>
       </xsd:complexType>
    </xsd:element>

     <xsd:element name="topicDataQosPolicy">
        <xsd:complexType>
          <xsd:all>
            <xsd:element name="value" type="xsd:base64Binary" minOccurs="1"
                              maxOccurs="1"/>
          </xsd:all>
        </xsd:complexType>
     </xsd:element>

    <xsd:element name="duration">
      <xsd:complexType>
        <xsd:all>
            <xsd:element name="sec" type="xsd:string" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="nanosec" type="xsd:string" minOccurs="1"
                              maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="durabilityQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element ref="durabilityKind" minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="durabilityKind">
     <xsd:simpleType>
        <xsd:restriction base="xsd:string">
```

```
            <xsd:enumeration value="VOLATILE_DURABILITY_QOS"/>
            <xsd:enumeration value="TRANSIENT_LOCAL_DURABILITY_QOS"/>
            <xsd:enumeration value="TRANSIENT_DURABILITY_QOS"/>
            <xsd:enumeration value="PERSISTENT_DURABILITY_QOS"/>
        </xsd:restriction>
     </xsd:simpleType>
    </xsd:element>

    <xsd:element name="durabilityServiceQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element ref="duration" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="historyKind" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="history_depth" type="xsd:positiveInteger"
                         minOccurs="1" maxOccurs="1"/>
            <xsd:element name="max_samples" type="xsd:positiveInteger"
                         minOccurs="1" maxOccurs="1"/>
            <xsd:element name="max_instances" type="xsd:positiveInteger"
                         minOccurs="1" maxOccurs="1"/>
            <xsd:element name="max_samples_per_instance"
                    type="xsd:positiveInteger" minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="historyKind">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="KEEP_LAST_HISTORY_QOS"/>
            <xsd:enumeration value="KEEP_ALL_HISTORY_QOS"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>

    <xsd:element name="latencyBudgetQosPolicy">
        <xsd:complexType>
          <xsd:all>
            <xsd:element ref="duration" minOccurs="1" maxOccurs="1"/>
          </xsd:all>
        </xsd:complexType>
    </xsd:element>
    <xsd:element name="livelinessQosPolicy">
        <xsd:complexType>
          <xsd:all>
            <xsd:element ref="duration" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="livelinessKind" minOccurs="1" maxOccurs="1"/>
          </xsd:all>
        </xsd:complexType>
    </xsd:element>

    <xsd:element name="reliabilityQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element ref="reliabilityKind" minOccurs="1" maxOccurs="1"/>
            <xsd:element ref="duration" minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="reliabilityKind">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="BEST_EFFORT_RELIABILITY_QOS"/>
            <xsd:enumeration value="RELIABLE_RELIABILITY_QOS"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>

    <xsd:element name="destinationOrderQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element ref="destinationOrderKind" minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="destinationOrderKind">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="BY_RECEPTION_TIMESTAMP_DESTINATIONORDER_QOS"/>
            <xsd:enumeration value="BY_SOURCE_TIMESTAMP_DESTINATIONORDER_QOS"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>

    <xsd:element name="livelinessKind">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="AUTOMATIC_LIVELINESS_QOS"/>
            <xsd:enumeration value="MANUAL_BY_PARTICIPANT_LIVELINESS_QOS"/>
            <xsd:enumeration value="MANUAL_BY_TOPIC_LIVELINESS_QOS"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>

    <xsd:element name="historyQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element ref="historyKind" minOccurs="1" maxOccurs="1"/>
            <xsd:element name="depth" type="xsd:positiveInteger" default="1"
                              minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="resourceLimitsQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element name="max_samples" type="xsd:positiveInteger"
                              minOccurs="1" maxOccurs="1"/>
            <xsd:element name="max_instances" type="xsd:positiveInteger"
                              minOccurs="1" maxOccurs="1"/>
            <xsd:element name="max_samples_per_instance"
                              type="xsd:positiveInteger" minOccurs="1"
                              maxOccurs="1"/>
            <xsd:element name="initial_samples" type="xsd:positiveInteger"
            minOccurs="1" maxOccurs="1"/>
            <xsd:element name="initial_instances" type="xsd:positiveInteger"
                              minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="transportPriorityQosPolicy">
      <xsd:complexType>
        <xsd:all>
```

```
            <xsd:element name="value" type="xsd:nonNegativeInteger"
                                 minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="lifespanQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element ref="duration" minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="ownershipQosPolicy">
      <xsd:complexType>
        <xsd:all>
            <xsd:element ref="ownershipKind" minOccurs="1" maxOccurs="1"/>
        </xsd:all>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="ownershipKind">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="SHARED_OWNERSHIP_QOS"/>
            <xsd:enumeration value="EXCLUSIVE_OWNERSHIP_QOS"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

9 Contacts & Notices

9.1 Contacts

PrismTech Corporation
400 TradeCenter
Suite 5900
Woburn, MA
01801
USA
Tel: +1 781 569 5819

PrismTech Limited
PrismTech House
5th Avenue Business Park
Gateshead
NE11 0NG
UK
Tel: +44 (0)191 497 9900

PrismTech France
28 rue Jean Rostand
91400 Orsay
France
Tel: +33 (1) 69 015354

Web: http://www.prismtech.com

E-mail: info@prismtech.com

9.2 Notices