RTI Message Service

Interoperability Guide

Version 5.2.3

Your systems. Working as one.

Trademarks

Copy and Use Restrictions

Third-Party Copyright Notices

Note: In this section, "the Software" refers to third-party software, portions of which are used in *RTI Message Service*; "the Software" does not refer to *RTI Message Service*.

Technical Support

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

Contents

Chapter 1 Introduction

This document describes the compatibility and interoperability between *RTI® Message Service* and other products. These products can be purchased separately from *RTI Message Service* itself, so every chapter of this guide will not apply to all customers. For more information about purchasing one of the products discussed in this guide, please contact your RTI account representative.

This document describes interoperability with:

- **DDS Applications**

  RTI Connext™ DDS supports the Data Distribution Service (DDS) specification from the Object Management Group (OMG). DDS incorporates both messaging and caching to deliver on the demanding requirements of the defense, financial services, industrial automation, and transportation industries. RTI's implementation of DDS supports a wide range of platforms and programming languages.

  Consider using DDS if your application requires greater control over data typing, filtering, and configuration than is possible with JMS or if your performance requirements can only be met by a native-language platform.

- **RTI Persistence Service**

  RTI Persistence Service enhances data availability by providing message persistence outside of any particular *RTI Message Service* or *RTI Connext DDS* application. Instances of this service can be deployed in redundant and/or load-balanced configurations to further reduce risk and CPU loads.

  Consider *RTI Persistence Service* if your distributed system requires a higher degree of availability than can be provided by a purely peer-to-peer messaging topology. See Chapter 3: RTI Persistence Service Interoperability.

- **RTI Secure WAN Transport**

  This network transport product supplements the built-in *RTI Message Service* transports—UDP/IPv4, UDP/IPv6, and shared memory—with two more transports that can be used separately or together:

  - A WAN transport, which allows *RTI Message Service*-based applications to communicate across NAT firewalls
  - A DTLS transport, which authenticates connections and encrypts messages on the network to provide high security

  Consider *RTI Secure WAN Transport* if you require secure, encrypted communication, possibly across a wide-area network. See Chapter 4: Secure WAN Transport Interoperability.

- WireShark, a network packet dissector. See Chapter 5: Interoperability with Wireshark.

❑ **Application Servers**

Managed applications running within an application server such as JBOSS Application Server or Oracle BEA WebLogic Server can use *RTI Message Service*, just like standalone applications can. If you are using JMS within the context of a Java web application or a broader SOA infrastructure, Chapter 6: Application Server Interoperability will be of interest to you.

❑ **Complex Event Processing**

Complex event processing (CEP) engines, such as Oracle CEP, provide extensive filtering and correlation capabilities. A CEP engine monitors incoming events from many sources and triggers outgoing events based on the patterns in those events and the correlations between them. CEP applications can use *RTI Message Service* to deliver input events to a CEP engine across the network and publish outgoing events to other applications. See Chapter 7: Complex Event Processing Interoperability.

Chapter 2 Interoperability with DDS Applications

RTI Message Service and RTI's implementation of DDS are both peer-to-peer, publish-subscribe middleware solutions. Although the former provides a message-centric solution and the latter a data-centric one, their capabilities are similar in many ways. In fact, it is possible for *RTI Message Service* and *Connext DDS* applications to communicate directly with one another in a peer-to-peer way, without any bridges or intermediaries.

This chapter will teach you how to achieve interoperability between these two products. It covers two aspects of this interoperability:

- Exchanging Messages (Section 2.1)
- Sharing Object Configurations (Section 2.2)

2.1 Exchanging Messages

One of the chief differences between *Message Service* and *Connext DDS* is that the JMS specification, implemented by the former, provides pre-defined **Message** types that applications use to communicate. In contrast, the DDS specification implemented by *Connext DDS* is based on an application's ability to directly expose its own data-types to the middleware. When these data types are described in OMG IDL, XML, or XSD, the *rtiddsgen* code generator can create type definitions in a variety of programming languages, including the serialization and deserialization code that allow the middleware to distribute instances of these types.

In order for communication to take place between two applications, they must agree on three things:

- They must use the same data type.
- They must use the same topic names.
- They must have compatible QoS parameters.

2.1.1 Data Types and Keys

Message Service allows you to associate a data type that you define with your topic definition, just like DDS does. This allows you to interoperate peer-to-peer with applications written to the OMG DDS API, regardless of what data types they use: simply associate that existing data type with your JMS topic, and *Message Service* will serialize and deserialize the messages on that topic exactly as a DDS implementation would. The on-the-network size of each message can also potentially be smaller this way than if you elect to use general-purpose JMS messages.

The following sections describe both interoperability options: using DDS-compatible data types or using general-purpose JMS message types.

2.1.1.1 Using DDS-Compatible Data Types

To associate your JMS topic with a DDS-compatible data type, do the following:

1. Define that data type in a format compatible with the *rtiddsgen* code-generation tool, such as XSD or OMG IDL, and generate Java code from that definition. If you have already done this for your DDS applications, you can use the code you have already generated. You can find instructions for using *rtiddsgen* in the *RTI Connext DDS Core Libraries Getting Started Guide* and *User's Manual*.
2. Place the JAR library containing your compiled generated code on the class path of your JMS application, and associate the topic definition in your *Message Service* configuration file with the type you want. This association consists of two XML attributes added to your topic definition:
   a. **class_name**: The fully qualified Java class name of your generated data-type.
   b. **registered_type_name**: The logical name under which you registered the data type in your DDS application (using the **TypeSupport.register_type()** method). Users often simply register their types under their physical names, either qualified or unqualified.

   For example:

```
<topic name="MyTopic"
    class_name="com.acme.MyType"
     registered_type_name="MyType">
    <!-- ... -->
</topic>
```

Objects of DDS-compatible, *rtiddsgen*-generated data types appear in JMS wrapped in **ObjectMessages**. Of course, *Message Service* will instantiate these objects and present them to you in **ObjectMessages** on the "receive side."Any data sample of a custom type that is published by a DDS-based application and received by a JMS-based application will appear as an **ObjectMessage** whose body consists of the published sample object. You must remember to do this yourself when sending messages from a JMS application. Custom topic data-types are incompatible with message types other than **ObjectMessage** and attempting to send an incompatible message will result in a sender-side error.

2.1.1.2 Using General-Purpose JMS Message Types

If you do not specify the two <topic> attributes described above, which associate your topic with a custom DDS-compatible data type, your topic will use a general-purpose on-the-network format to communicate with DDS applications. This format allows you to use any of the JMS-standard message types. It also allows you to interoperate with DDS-based applications. However, those applications must use the type definitions provided by *Message Service*, or you must deploy an instance of *RTI Routing Service* to transform the *Message Service* message format to and/or from the data types used by the DDS-based application.

Your *Message Service* distribution includes OMG IDL type-definitions that correspond to the JMS message types. These types can be found in the file **RTIJMSHOME/resource/idl/jms_message.idl**. To allow a DDS application to communicate with a *Message Service*-based one, simply treat this IDL file as you would one of your own: use the *rtiddsgen* tool to generate code in the language of your code, and then use the data types it defines in your application as you would any other type.

The data type that corresponds to all of the JMS message types, which is defined in this file, is **jms_message.idl**. This type supports both keyed and unkeyed data, depending on how you generate the code for it.

Generate unkeyed type definitions:

```
> rtiddsgen -DUNKEYED jms_message.idl
```

Generate keyed type definitions:

```
> rtiddsgen -DKEYED jms_message.idl
```

As with any data type, a DDS application must register this type under a logical name before it can be used. This name differs based on whether the type was generated as keyed or unkeyed, and is defined in the IDL file. These two type names are UNKEYED_TYPE_NAME and KEYED_TYPE_NAME. The following example code shows how to use these symbolic names:

Register unkeyed type in C++:

```
JmsMessageTypeSupport::register_type(   myDomainParticipant,
                                        UNKEYED_TYPE_NAME);
```

Register keyed type in Java:

```
JmsMessageTypeSupport.register_type(    myDomainParticipant,
                                        KEYED_TYPE_NAME.VALUE);
```

2.1.2 Topic Names

In both products, topics are defined based on application-specified names. In order for applications to communicate, they must agree on these names.

Creating a Topic in RTI Message Service:

```
Topic myTopic = (Topic) myInitialContext.lookup("My Topic Name");
```

Creating a Topic in DDS:

```
Topic myTopic = myDomainParticipant.create_topic(
    "My Topic Name",
    UNKEYED_TYPE_NAME.VALUE,            // or KEYED_TYPE_NAME.VALUE
    TOPIC_QOS_DEFAULT,                  // or other QoS
    null,                               // or non-null listener
    StatusKind.STATUS_MASK_NONE);       // or listener mask
```

A DDS application must pass some additional arguments to the **create_topic** method; these are documented in the manual for that product.

2.1.3 Quality of Service

In order for two applications to communicate, whether they use the JMS or DDS API, they must have compatible QoS specifications. The same QoS policies are available in both products; these policies, and their respective compatibility rules, are described in the documentation for these products and are not repeated here.

Applications can describe QoS configurations either declaratively, in XML files, or in application code. If these configurations are described in XML, they can be shared—with some limitations—between *Message Service* and DDS applications. Read on for more information.

2.2 Sharing Object Configurations

Both JMS and DDS applications can describe QoS policies declaratively in XML files. The formats of these files, however, are not identical; each format is aligned with the API and concepts of their respective products. In order to share configuration information between the two products, you will need to understand the differences between the two XML formats and how to transform from one to the other.

2.2.1 XML File Formats

The two file formats, although they are both XML-based, have different grammars. The format differences generally fall into one of two categories: (1) differences in terminology and (2) differences in structure.

2.2.1.1 Terminology Differences

The XML elements used in each file are named according to the classes in the APIs of each product. These names are different in JMS and DDS, but they have an almost one-to-one correspondence, as you can see in the following table.

JMS	DDS
ConnectionFactory	DomainParticipantFactory
Connection	DomainParticipant
Session	Publisher, Subscriber
Destination/Topic	Topic
MessageProducer	DataWriter
MessageConsumer	DataReader

For example, the XML element **producer_resource_limits** in a *Message Service* configuration file is called **writer_resource_limits** in a DDS file.

2.2.1.2 Structural Differences

JMS defines two types of administered objects: destinations and connection factories. Therefore, all objects in a *Message Service* configuration file are rooted in one of these. For example, *MessageProducer* QoS are specified within the context of the *Topic* with which that producer is associated:

```
<topic name="Example Topic">
    <producer_defaults>
           <!-- MessageProducer QoS here -->
    </producer_defaults>
</topic>
```

In DDS, QoS configurations for all types of objects are peers of one another:

```
<datawriter_qos name="Example Writer QoS">
    <!-- DataWriter QoS here -->
</datawriter_qos>
<datareader_qos name="Example Reader QoS">
    <!-- DataReader QoS here -->
</datareader_qos>
```

2.2.2 Automated File Transformation

Your *Message Service* distribution contains a script that can transform a configuration file from the JMS-centric file format to an equivalent file in the DDS-centric file format. (Because of the Structural Differences (Section 2.2.1.2), the reverse translation is not possible.) This script is called **jms2dds**, and is in the **bin/** directory of your installation. It has a very simple invocation syntax:

```
> jms2dds input_jms_file.xml output_dds_file.xml
```

For example:

```
> jms2dds /home/me/myJmsFile.xml /home/you/soonToBeCreatedDdsFile.xml
```

The resulting file in DDS format can also be used with other RTI technologies, such as the *RTI Persistence Service*. See Chapter 3: RTI Persistence Service Interoperability for more information about that service.

Chapter 3 RTI Persistence Service Interoperability

RTI Persistence Service is an *RTI Message Service* application that saves messages to transient or permanent storage, so they can be delivered to subscribing applications that join the system at a later time—even if the publishing application has already terminated.

This chapter assumes you have a basic understanding of terms such as *Connections*, *MessageProducers*, *Topics*, and Quality of Service (QoS) policies. These objects are described in the *RTI Message Service User's Manual*. You should especially have read Chapter 7, "Scalable High-Performance Applications: Durability and Persistence for High Availability," in the User's Manual.

3.1 Introduction

RTI Persistence Service runs as a separate application; you can run it on the same node as the publishing application, the subscribing application, or some other node in the network.

When configured to run in PERSISTENT mode, *RTI Persistence Service* stores its data using an ODBC database. For each persistent topic, it collects all the messages written by the corresponding persistent *MessageProducers* and stores them in topic tables in the database. Thus, multiple topics can be persisted through the same database. See the *RTI Persistence Service Getting Started Guide* for more information about the supported databases and how to configure them.

A relational database is not required if *RTI Persistence Service* is used only in TRANSIENT mode.

3.2 Configuring RTI Persistence Service

To use *RTI Persistence Service*:

1. Modify your *RTI Message Service* applications.

- The Durability QoS policy controls whether or not, and how, published messages are stored by *RTI Persistence Service* for delivery to late-joining *MessageConsumers*. See Chapter 7, "Scalable High-Performance Applications: Durability and Persistence for High Availability," in the User's Manual for more information.
    - For each *MessageProducer* whose data must be stored, set the Durability QoS policy's *kind* to PERSISTENT_DURABILITY_QOS or TRANSIENT_DURABILITY_QOS.
    - For each *MessageConsumer* that needs to receive stored data, set the Durability QoS policy's *kind* to PERSISTENT_DURABILITY_QOS or TRANSIENT_DURABILITY_QOS.
- Optionally, modify the Durability Service QoS Policy (Section 3.2.6.2), which can be used to configure *RTI Persistence Service*.

By default, the History and Resource Limits QoS policies used by the Persistence Service with topic 'A' will be configured using the Durability Service QoS policy of the first-discovered *MessageProducer* publishing 'A'. These values will overwrite the values specified in the XML file (unless you use the tag **<use_durability_service>** in the persistence group definition, see Durability Service QoS Policy (Section 3.2.6.2)).

2. Create a configuration file, as described in XML Configuration File (Section 3.2.1).
3. .Start *RTI Persistence Service* with your configuration file, as described in Starting RTI Persistence Service (Section 3.3.1). You can start it on either application's node, or even an entirely different node (provided that node is included in one of the applications' *initial_peers* lists).

3.2.1 XML Configuration File

The configuration file uses the DDS XML format. You can either write the file in that format from scratch, or you can create the file in the *Message Service* format and use the **jms2dds** script from your *Message Service* distribution to transform it. For more information about the DDS file format, see Chapter 2: Interoperability with DDS Applications and that product's documentation, if you are a customer.

Let's look at a very basic configuration file, just to get an idea of its contents. You will learn the meaning of each line as you read the rest of this section:

- ❏ QoS Configuration (Section 3.2.2)
- ❏ Configuring the RTI Persistence Service Application (Section 3.2.3)
- ❏ Configuring the Database Connection (Section 3.2.4)
- ❏ Configuring Connections (Section 3.2.5)
- ❏ Creating Persistence Groups (Section 3.2.6)

Example Configuration File

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!-- A Configuration file may be used by several
     persistence services specifying multiple
     <persistence_service> entries
-->
<dds>
    <!- QoS LIBRARY SECTION -->
    <qos_library name="QosLib1">
        <qos_profile name="QosProfile1">
            <writer_qos name="WriterQos1">
```

```
                    <history>
                        <kind>KEEP_ALL_HISTORY_QOS</kind>
                    </history>
                </writer_qos>
                <reader_qos name="ReaderQos1">
                    <reliability>
                        <kind>RELIABLE_RELIABILITY_QOS</kind>
                    </reliability>
                    <history>
                        <kind>KEEP_ALL_HISTORY_QOS</kind>
                    </history>
                </reader_qos>
            </qos_profile>
        </qos_library>

        <!-PERSISTENCE SERVICE SECTION -->
        <persistence_service name="Srv1">

            <!-DATABASE CONNECTION SECTION -->
            <database_connection>
                <dsn>MyDSN</dsn>
                <username>MyUsername</username>
                <password>MyPassword</password>
                <restore>1</restore>
            </database_connection>

            <!-DOMAIN PARTICIPANT SECTION -->
            <participant name="Part1">
                <domain_id>71</domain_id>

                <!-PERSISTENCE GROUP SECTION -->
                <persistence_group name="PerGroup1" filter="*">
                    <single_publisher>true</single_publisher>
                    <single_subscriber>true</single_subscriber>
                    <writer_qos base_name="QosLib1::QosProfile1"/>
                    <reader_qos base_name="QosLib1::QosProfile1"/>
                </persistence_group>

            </participant>
        </persistence_service>
    </dds>
```

3.2.1.1 Configuration File Syntax

The XML configuration file must follow these syntax rules:

- ❏ All values are case-sensitive unless otherwise stated.
- ❏ The root tag of the configuration file must be <dds> and end with </dds>.
- ❏ The primitive types for tag values are specified in Table 3.1.

Table 3.1 **Supported Tag Values**

Type	Format	Notes
Boolean	yes, 1, true, BOOLEAN_TRUE: these all mean TRUE	Not case-sensitive
	no, 0, false, BOOLEAN_FALSE: these all mean FALSE	
Enum	A string. Legal values are described in the *User's Manual*.	Must be specified as a string.
Long	-2147483648 to 2147483647 or 0x80000000 to 0x7fffffff or LENGTH_UNLIMITED	A 32-bit signed integer
UnsignedLong	0 to 4294967296 or 0 to 0xffffffff	A 32-bit unsigned integer
String	UTF-8 character string	All leading and trailing spaces are ignored between two tags

3.2.2 QoS Configuration

Each persistence group and connection has a set of QoSs. There are six tags:

- <participant_qos>
- <publisher_qos>
- <subscriber_qos>
- <topic_qos>
- <writer_qos>
- <reader_qos>

Each QoS is identified by a name. The QoS can inherit its values from other QoSs described in the XML file. For example:

```
<writer_qos name="DerivedProducerQos" base_qos_name="Lib::BaseProducerQos">
    <history>
        <kind>KEEP_ALL_HISTORY_QOS</kind>
    </history>
</writer_qos>
```

In the above example, the producer QoS named 'DerivedProducerQos' inherits the values from the producer QoS 'BaseProducerQos' contained in the library 'Lib'. The History QoS policy **kind** is set to KEEP_ALL_HISTORY_QOS.

The persistence groups and connections can use QoS libraries and profiles to configure their QoS values. For example:

```
<dds>
    <!- QoS LIBRARY SECTION -->
    <qos_library name="QosLib1">
```

```
            <qos_profile name="QosProfile1">
                <writer_qos name="WriterQos1">
                    <history>
                        <kind>DDS_KEEP_ALL_HISTORY_QOS</kind>
                    </history>
                </writer_qos>
            </qos_profile>
        </qos_library>

        <!-PERSISTENCE SERVICE SECTION -->
        <persistence_service name="Srv1">
            …
            …
            …
            <!-PERSISTENCE GROUP SECTION -->
            <persistence_group name="PerGroup1" filter="*">
                <single_publisher>true</single_publisher>
                <single_subscriber>true</single_subscriber>
                <writer_qos base_name="QosLib1::QosProfile1"/>
           </persistence_group>
        </persistence_service>
    </dds >
```

3.2.3 Configuring the RTI Persistence Service Application

Each execution of the *RTI Persistence Service* application is configured using the content of a persistence service application tag (<persistence_service>). When *RTI Persistence Service* is started, you can specify which <persistence_service> tag to use for configuring the service.

For example:

```
<dds>
    <persistence_service name="Srv1">
    …
    </persistence_service>
</dds>
```

If you do not specify a service name when you start *RTI Persistence Service* (as described in Starting RTI Persistence Service (Section 3.3.1)), the name in the first <persistence_service> tag in the file will be used.

Because a configuration file may contain multiple <persistence_service> tags, one file can be used to configure multiple *RTI Persistence Service* executions.

Table 3.2 lists the tags you can specify for a persistence service. For default values, please see the online (HTML) documentation.

Table 3.2 **Persistence Service Application Tags**

Tag	Description	Number of Tags Allowed
<database_connection>	Database connection used to persist the topic data.	0 or 1
<participant>	The participant tag describes a Connection created by the persistence service to monitor a domain ID. At least one participant tag should be specified.	1 or more
<synchronize>	A Boolean (see Table 3.1, "Supported Tag Values," on page 3-4) that indicates if redundant persistence service instances should synchronize their states with one another. When set to TRUE, messages lost on the way to one service instance can be repaired by another without impacting the original publisher of that message. To synchronize the instances, the tag <synchronize> must be set to true in every instance involved in the synchronization. Note: This synchronization mechanism is not equivalent to database replication. The extent to which database instances have identical contents depends on the destination ordering and other QoS settings for the persistence service instances.	0 or 1

3.2.4 Configuring the Database Connection

The <database_connection> tag is used to associate a database to the persistence service. If the <database_connection> tag is not specified, the persistence will operate in TRANSIENT mode and all the data will be kept in memory. Otherwise, the persistence service will operate in PERSISTENT mode and all the topic data will be stored into the database whose DSN (Data Source Name) is specified in the <dsn> tag. For example:

```
<persistence_service name="Srv1">
    <database_connection>
        <dsn>MyDSN</dsn>
        <username>MyUsername</username>
        <password>MyPassword</password>
        <restore>1</restore>
    </database_connection>
</persistence_service>
```

Table 3.3 further describes the <database_connection> tags. For default values, please see the online (HTML) documentation.

3.2.5 Configuring Connections

An XML <persistence_service> tag will contain a set of connections. The persistence service will persist topics published in the domain IDs associated with these connections. For example:

```
<persistence_service name="Srv1">
    <participant name="Part1">
        <domain_id>71</domain_id>
        ...
    </participant>
    <participant name="Part2">
```

Table 3.3 **Database Connection Tags**

Tag	Description	Number of Tags Allowed
<dsn>	DSN used to connect to the database using ODBC. You should create this DSN through the ODBC settings on Windows systems, or in your .odbc.ini file on UNIX/Linux systems. This tag is required.	1
<username>	Username to connect to the database.	0 or 1
<password>	Password to connect to the database.	0 or 1
<restore>	A Boolean (see Table 3.1, "Supported Tag Values," on page 3-4) that indicates if the topic data associated with a previous execution of the persistence service must be restored or not. If the topic data is not restored, it will be deleted from the database.	0 or 1
<odbc_library>	Specifies the ODBC driver to load. By default, *Message Service* will try to use the standard ODBC driver manager library (UnixOdbc on UNIX/Linux systems, the Windows ODBC driver manager on Windows systems).	0 or 1

```
            <domain_id>72</domain_id>
            ...
        </participant>
    </persistence_service>
```

Using the above example, the persistence service will create two connections on domains 71 and 72. After the connections are created, the persistence service will monitor the discovery traffic looking for topics to persist.

The <domain_id> tag can be specified alternatively as an attribute of <participant>. For example:

```
<persistence_service name="Srv1">
    <participant name="Part1" domain_id="71">
        ...
    </participant>
</persistence_service>
```

Table 3.4, "Participant Tags," on page 3-7 further describes the participant tags.

Table 3.4 **Participant Tags**

Tag	Description	Number of Tags Allowed
<domain_id>	Domain ID associated with the Connection. The domain ID can be specified as an attribute of the connection tag.	0 or 1
<participant_qos>	Connection QoS.	0 or 1
<persistence_group>	A persistence group describes a set of topics whose data that must be persisted by the persistence service.	1 or more

3.2.6 Creating Persistence Groups

The topics that must be persisted in a specific domain ID are specified using <persistence_group> tags. A <persistence_group> tag defines a set of topics identified by a POSIX expression.

For example:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="H*">
        ...
    </persistence_group>
</participant>
```

In the previous example, the persistence group 'PerGroup1' is associated with all the topics published in domain 71 whose name starts with 'H'.

<participant> tag can contain multiple persistence groups; the set of topics that each one represents can intersect.

Table 3.5, "Persistence Group Tags," on page 3-8 further describes the persistence group tags. For default values, please see the online (HTML) documentation.

Table 3.5 **Persistence Group Tags**

Tag	Description	Number of Tags Allowed
<filter>	A list of POSIX expressions separated by commas that describe the set of topics associated with the persistence group. The filter can be specified as an attribute of <persistence_group> as well.	0 or 1
<content_filter>	Content filter topic expression. A persistence group can subscribe to a specific set of data based on the value of this expression. A filter expression is similar to the WHERE clause in SQL. For more information on the syntax, please see Chapter 5, "Subscribing to Messages," in the User's Manual.	0 or 1
<propagate_dispose>	A Boolean (see Table 3.1) that controls whether or not the persistence service propagates dispose messages from MessageProducers to MessageConsumers.	0 or 1
<propagate_unregister>	A Boolean (see Table 3.1) that controls whether or not the persistence service propagates unregister messages from MessageProducers to MessageConsumers.	0 or 1
<single_publisher>	A Boolean (see Table 3.1) that indicates if the persistence service should create one Session per persistence group or one Session per MessageProducer inside the persistence group.	0 or 1
<single_subscriber>	A Boolean (see Table 3.1) that indicates if the persistence service should create one Session per persistence group or one Session per MessageConsumer in the persistence group.	0 or 1

Table 3.5 **Persistence Group Tags**

Tag	Description	Number of Tags Allowed
<use_durability_service>	A Boolean (see Table 3.1) that indicates if the HISTORY and RESOURCE_LIMITS QoS policy of the MessageProducers and MessageConsumers should be configured based on the DURABILITY SERVICE value of the discovered MessageProducers.	0 or 1
<share_database_connection>	A Boolean (see Table 3.1) that indicates if the persistence group must use a separate ODBC database connection to store the topic data received by each MessageConsumer.	0 or 1
<consumer_checkpoint_frequency>	This property controls how often (expressed as a number of messages) the MessageConsumer state is stored in the database. The MessageConsumers are the MessageConsumers created by the persistence service. A high frequency will provide better performance. However, if the persistence service is restarted, it may receive some duplicate messages. The persistence service will send these duplicates messages on the wire but they will be filtered by the MessageConsumers and they will not be propagated to the application. This property is only applicable when the persistence service operates in persistent mode (the <database_connection> tag is present).	0 or 1
<producer_in_memory_state>	A Boolean (see Table 3.1) that determines how much state will be kept in memory by the MessageProducers in order to avoid accessing the database. The property is only applicable when the persistence service operates in persistent mode (the <database_connection> tag is present). If this property is TRUE, the MessageProducers will keep a copy of all the instances in memory. They will also keep a fixed state overhead of 24 bytes per message. This mode provides the best performance. However, the restore operation will be slower and the maximum number of messages that a MessageProducer can manage will be limited by the available physical memory. If this property is FALSE, all the state will be kept in the underlying database. In this mode, the maximum number of messages that a MessageProducer can manage will not be limited by the available physical memory.	0 or 1
<topic_qos>	Topic QoS	0 or 1
<publisher_qos>	Publisher QoS	0 or 1
<subscriber_qos>	Subscriber QoS	0 or 1
<writer_qos>	MessageProducer QoS[a]	0 or 1
<reader_qos>	MessageConsumer QoS	0 or 1

a. These fields cannot be set and are assigned automatically: protocol.virtual_guid, protocol.rtps_object_id, durability.kind.

3.2.6.1 QoSs

When a persistence service discovers a topic 'A' that matches a specific persistence group, it creates a consumer and producer to persist that topic. The QoSs associated with these consumers and producers, as well as the corresponding Sessions, can be configured inside the persistence group using QoS tags.

For example:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="*">
     ...
        <publisher_qos base_qos_name="QosLib1::PubQos1"/>
        <subscriber_qos base_qos_name="QosLib1::SubQos1"/>
        <writer_qos base_qos_name="QosLib1::WriterQos1"/>
        <reader_qos base_qos_name="QosLib1::ReaderQos1"/>
    ...
    </persistence_group>
</participant>
```

For instance, the number of messages saved by *RTI Persistence Service* is configurable through the HISTORY QoS policy of the MessageProducers.

If a QoS tag is not specified the persistence service will use the corresponding *Message Service* default values (Durability Service QoS Policy (Section 3.2.6.2) describes an exception to this rule).

3.2.6.2 Durability Service QoS Policy

The DURABILITY SERVICE QoS policy associated with a *MessageProducer* is used to configure the HISTORY and the RESOURCE_LIMITS associated with the MessageConsumers and MessageProducers.

By default, the HISTORY and RESOURCE_LIMITS of a MessageConsumer and *MessageProducer* with topic 'A' will be configured using the DURABILITY_SERVICE value of the first discovered *MessageProducer* publishing 'A'. These values will overwrite the values specified in the XML file.

To not overwrite the XML values, you can use the tag <use_durability_service> in the persistence group definition:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="*">
    ...
        <use_durability_service/>0</ use_durability_service>
    ...
    </persistence_group>
</participant>
```

3.2.6.3 Sharing a Session

By default, the MessageProducers and MessageConsumers associated with a persistence group will share the same Session.

To associate a different Session with each MessageProducer and MessageConsumer, use the tags <single_publisher> and <single_subscriber>, as follows:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="*">
    ...
```

```
            <single_publisher/>0</single_publisher>
            <single_subscriber/>0</single_subscriber>
        ...
        </persistence_group>
    </participant>
```

3.2.6.4 Sharing a Database Connection

By default, the persistence service will share a single ODBC database connection to persist the topic data received by each MessageConsumer.

To associate an independent database connection to the MessageConsumers created by the persistence service, use the tag <share_database_connection>, as follows:

```
<participant name="Part1">
    <domain_id>71</domain_id>
    <persistence_group name="PerGroup1" filter="*">
    ...
        <share_database_connection>0</share_database_connection>
    ...
    </persistence_group>
</participant>
```

Sharing a database connection optimizes the resource usage. However, the concurrency of the system decreases because the access to the database connection must be protected.

3.3 Running RTI Persistence Service

You can run *RTI Persistence Service* on any node in the network. It does not have to be run on the same node as the publishing or subscribing applications for which it is saving/delivering data. If you run it on a separate node, however, make sure that the other applications can find it during the discovery process—that is, it must be in one of the initial_peers lists.

3.3.1 Starting RTI Persistence Service

RTI Persistence Service's executable is located in **$RTIJMSHOME/bin**.

```
$ rtipersistenceservice -cfgFile <xml_configuration_file>
[-srvName <SERVICENAME>] [-serviceVerbosity <0-3>]
[-ddsVerbosity <0-3>] [-help]
```

You must specify a configuration file. Configuring RTI Persistence Service (Section 3.2) describes the contents of the file.

Other optional arguments are described in Table 3.6.

Table 3.6 **Command-Line Options**

Command-line Option	Description
-cfgFile	This option specifies the XML configuration file for the persistence service. *This option is required.*
-help	Prints version information and list of command-line options.

Table 3.6 **Command-Line Options**

Command-line Option	Description
-srvName	This option is used to configure the persistence service name. The same configuration file can be used to configure multiple persistence services. Each persistence service will load its configuration from a different <persistence_service> tag based on the name specified with this command line parameter. If srvName is not specified, the persistence service will be configured using the first <persistence_service> tag in the configuration file.
-serviceVerbosity	*RTI Persistence Service* verbosity: 0 - No verbosity 1 - Exceptions (default) 2 - Exceptions and warnings 3 - Exceptions, warnings and status information
-ddsVerbosity	*Message Service*/DDS verbosity: 0 - No verbosity 1 - Exceptions (default) 2 - Exceptions and warnings 3 - Exceptions, warnings and status information

3.3.2 Stopping RTI Persistence Service

To stop RTI Persistence Service: press **Ctrl-C**. *RTI Persistence Service* will close all files and perform a clean shutdown.

Chapter 4 Secure WAN Transport Interoperability

Secure WAN Transport provides transport plugins that can be used by developers of *Message Service* applications. These transport plugins allow *Message Service* applications running on private networks to communicate securely over a Wide-Area Network (WAN), such the internet. There are two primary components in the package which may be used independently or together: communication over Wide-Area Networks that involve Network Address Translators (NATs), and secure communication with support for peer authentication and encrypted data transport.

4.1 Introduction

The *Message Service* core is transport-agnostic. *Message Service* offers three built-in transports: UDP/IPv4, UDP/IPv6, and inter-process shared memory. The implementation of NAT traversal and secure communication is done at the transport level so that your messages application is not affected and does not need to be changed, although there is additional on-the-wire traffic.

The basic problem to overcome in a WAN environment is that messages sent from an application on a private local-area network (LAN) appear to come from the LAN's router address, not from the internal IP address of the host running the application. This is due to the existence of a Network Address Translator (NAT) at the gateway. This does not cause problems for client/server systems because only the server needs to be globally addressable; it is only a problem for systems with peer-to-peer communication models, such as *Message Service*. *Secure WAN Transport* solves this problem, allowing communication between peers that are in separate LAN networks, using a UDP hole-punching mechanism based on the STUN protocol (IETF RFC 3489bis) for NAT traversal. This requires the use of an additional rendezvous server application, the RTI WAN Server.

Once the transport has enabled traffic to cross the NAT gateway to the WAN, it is flowing on network hardware that is shared (in some cases, over the public internet). In this context, it is important to consider the security of data transmission. There are three primary issues involved:

- authenticating the communication peer (source or destination) as a trusted partner;
- encrypting the data to hide it from other parties that may have access to the network;
- validating the received data to ensure that it was not modified in transmission.

Secure WAN Transport addresses these problems by wrapping all RTPS-encoded data using the DTLS protocol (IETF RFC 4347), which is a variant of SSL/TLS that can be used over a datagram network-layer transport such as UDP. The security features of the WAN Transport may also be used on an untrusted local-area network with the Secure Transport.

In summary, the package includes two transports:

- The WAN Transport is for use on a WAN and includes security. It must be used with the WAN Server, a rendezvous server that provides the ability to discover public addresses and to register and look up peer addresses based on a unique WAN ID. The WAN Server is based on the STUN (Session Traversal Utilities for NAT) protocol [draft-ietfbehave-rfc3489bis], with some extensions. Once information about public addresses for the application and its peers has been obtained and connections have been initiated, the server is no longer required to maintain communication with a peer. (Note: security is disabled by default.)
- The Secure Transport is an alternate transport that provides security on an untrusted LAN. Use of the RTI WAN Server is not required.

Multicast communication is not supported by either of these transports.

This section provides a technical overview of:

- WAN Traversal via UDP Hole-punching (Section 4.1.1)
- WAN Locators (Section 4.1.2)
- Datagram Transport-Layer Security (DTLS) (Section 4.1.3)
- Certificate Support (Section 4.1.4)

For information on how to use *Secure WAN Transport* with your *Message Service* application, see Chapter 25: Configuring RTI Secure WAN Transport.

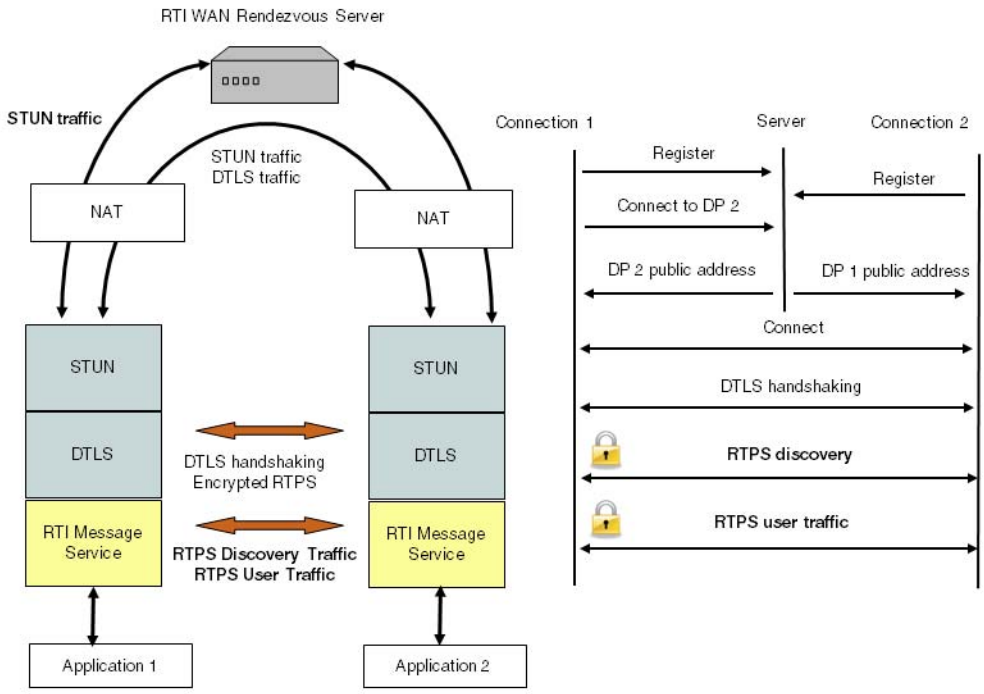
4.1.1 WAN Traversal via UDP Hole-punching

In order to resolve the problem of communication across NAT boundaries, the WAN Transport implements a UDP hole-punching solution for NAT traversal [draft-ietf-behave-p2p-state]. This solution uses a rendezvous server, which provides the ability to discover public addresses, and to register and lookup peer addresses based on a unique WAN ID. This server is based on the STUN (Session Traversal Utilities for NAT) protocol [draft-ietf-behave-rfc3489bis], with some extensions. This protocol is a part of the solution used for standards-based voice over IP applications; similar technology has been used by systems such as Skype and has proven to be highly reliable. A key advantage of STUN is that it is based on UDP and therefore is able to preserve the real-time characteristics of the DDS Interoperability Wire Protocol.

Once information about public addresses for the application and its peers has been obtained, and connections have been initiated, the server is no longer required to maintain communication with a peer. However, if communication fails, possibly due to changes in dynamically-allocated addresses, the server will be needed to reopen new public channels.

Figure 4.1 shows the RTI WAN transport architecture.

Figure 4.1 **RTI WAN Transport Architecture**



4.1.1.1 Protocol Details

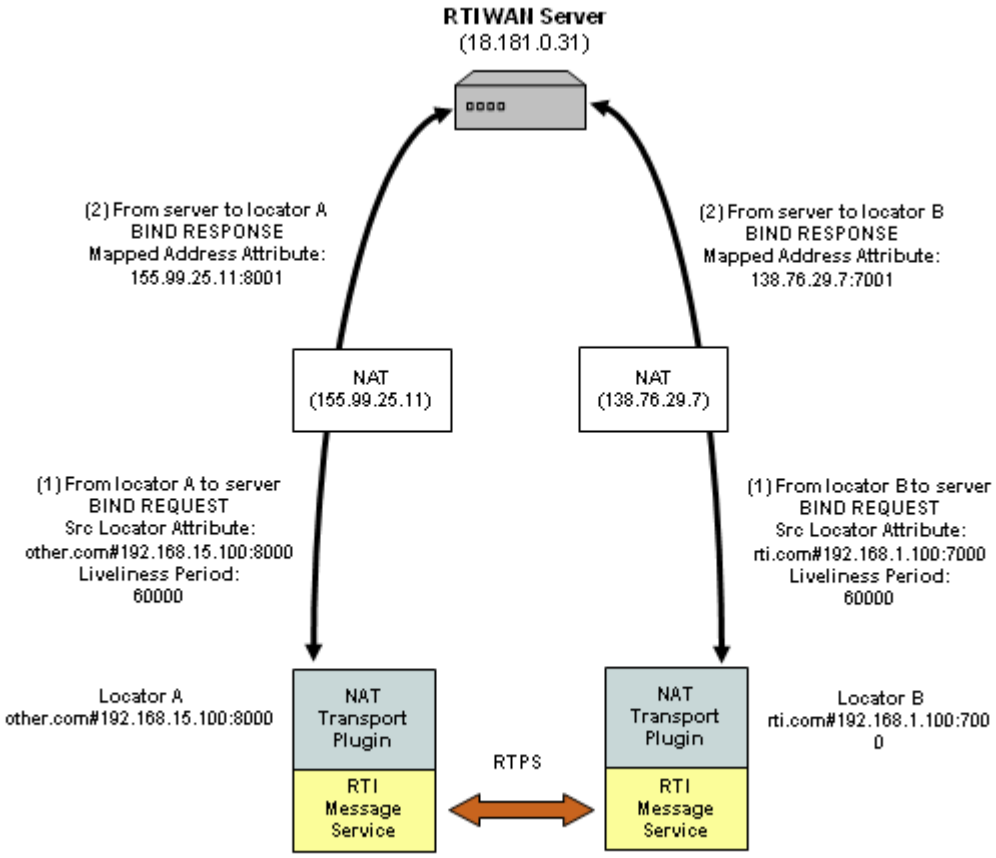
The UDP hole-punching algorithm implemented by the WAN transport has two different phases: registration and connection. This algorithm only works with cone or asymmetric NATs where the same public address/port is assigned to all the sessions with the same private address/port address.

- Registration Phase

  The RTI WAN Server application runs on a machine that resides on the WAN network (i.e., not in a private LAN). It has to be globally accessible to LAN applications. It is started by a script and acts as a rendezvous point for LAN applications. During the registration phase, each transport locator is registered with the RTI WAN Server using a STUN binding request message.

The RTI WAN Server associates RTPS locators with their corresponding public IPv4 transport addresses (a combination of IP address and port) and stores that information in an internal table. Figure 4.2 illustrates the registration phase.

Figure 4.2 **Registration Phase**



❑ Connection Phase

The connection phase starts when locator A wants to establish a connection with locator B. Locator A obtains information about locator B via *Message Service* discovery traffic or the **initial_peers** list. To establish a connection with locator B, locator A sends a STUN connect request to the RTI WAN server. The server sends a STUN connect response to locator A, including information about the public IP transport address (IP address and port) of locator B. In parallel, the RTI WAN server contacts locator B using another STUN connect request to let it know that locator A wants to establish a connection with it.

When locator A receives the public IP address of locator B, it will try to contact B using two STUN binding request messages. The first message is sent to the public address of B and the second message is sent to the private address of B. The private address was obtained using the last 32 bits of the locator address of B. The STUN binding request message directed to the public transport address of B sent by locator A will open a hole in A's NAT to receive messages from B.

When locator B receives the public address of locator A, it will try to contact A sending a STUN binding request message to that public address. This message will open a hole in B's NAT to receive messages from A. When locator A receives the first STUN binding response from locator B, it starts sending RTPS traffic.

The connection phase includes two processes: the connect process (Figure 4.3 on page 4-5) and the NAT hole punching process (Figure 4.4 on page 4-6).

Figure 4.3 **Connect Process**

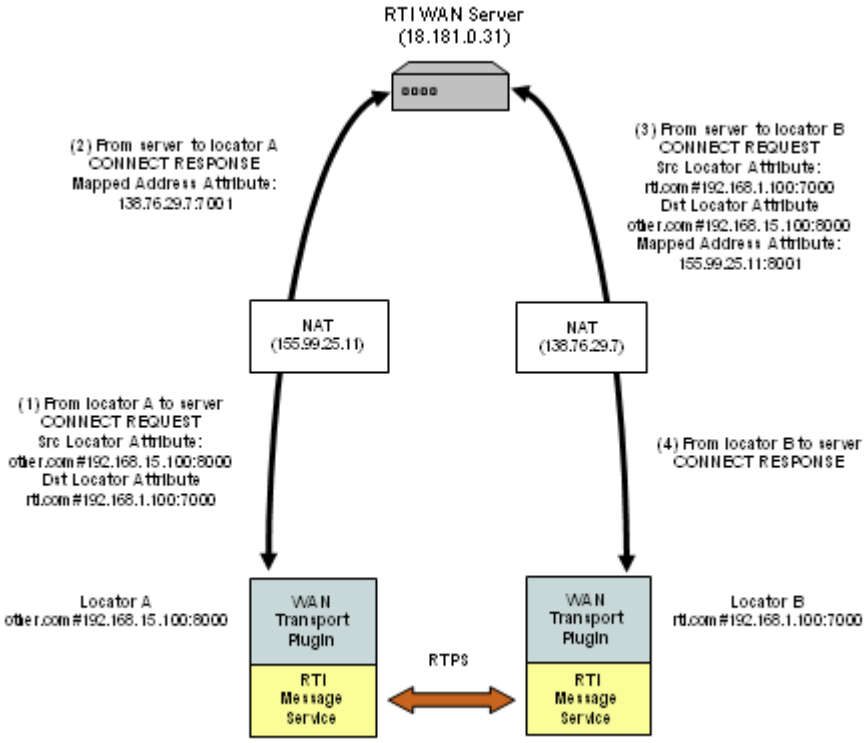
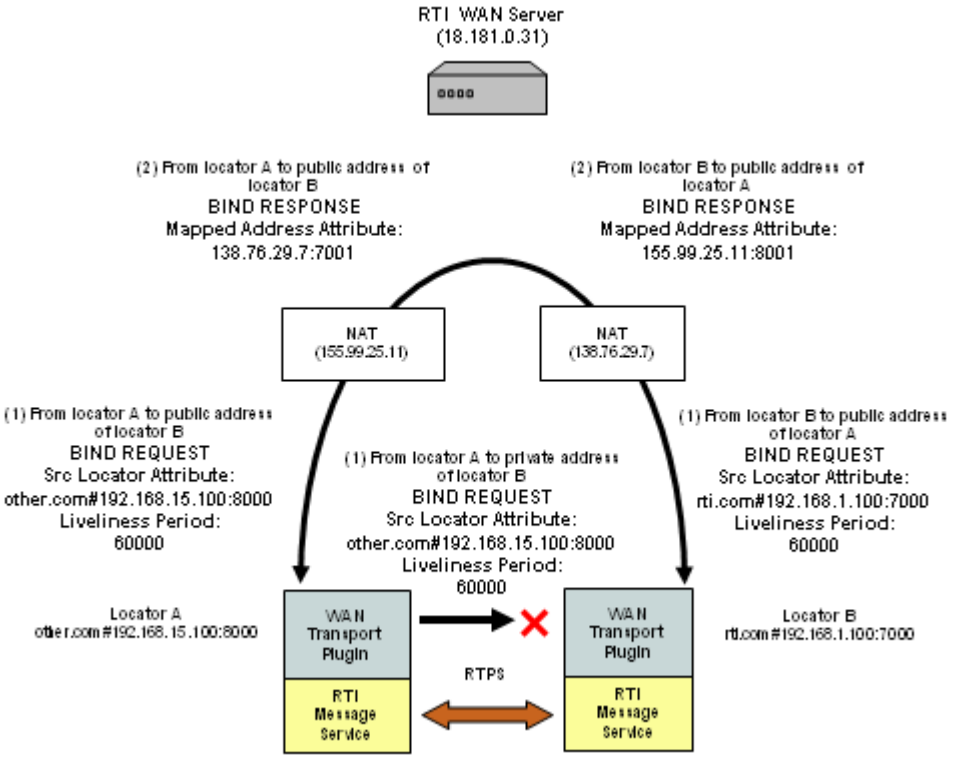


Figure 4.4 **NAT Hole Punching Process**



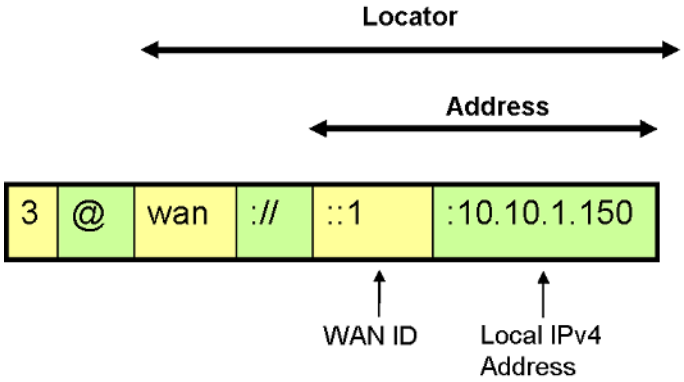
❑ STUN Liveliness

Finally, since bindings allocated by NAT expire unless refreshed, the clients (locators) must generate binding request messages for the server and other clients to refresh the bindings. The RTI STUN protocol implementation uses the attribute LIVELINESS-PERIOD in the STUN binding request to indicate the period in milliseconds at which a client will assert its liveliness. The WAN Server will remove a locator from its mapping table when the liveliness contract is not met. Likewise, a transport instance will remove a STUN connection with a locator when this locator does not assert its liveliness as indicated in the last binding request.

4.1.2 WAN Locators

The WAN transport does not use simple IP addresses to locate peers. A WAN transport locator consists of a WAN ID, which is an arbitrary 12-byte value, and a bottom 4-byte value that speci-

fies a fallback local IPv4 address. Your peers list (**initial_peers**) must be configured to look for peers with locators of the form:



- The address is a 128-bit address in IPv6 notation.
- The "wan://" part specifies that the address is for the WAN transport.
- The next part, "::1", specifies the top 12 bytes of the address to be 11 zero bytes, followed by a byte with value 1 (this corresponds to the peer's WAN ID).
- The last part, "10.10.1.150" refers to the peers local IPv4 address, which will be used if the peers are on the same local network.

A *Connection* using the WAN transport will have to initialize the Discovery QoS policy's **initial_peers** with the WAN locator addresses corresponding to the peers to which it wants to connect. The value of the Discovery QoS policy's **initial_peers** can be set using the environment variable initial_peers or the initial_peers configuration file.

4.1.3 Datagram Transport-Layer Security (DTLS)

Data security is provided by wrapping all messaging network traffic with the Datagram Transport Layer Security (DTLS) protocol (IETF RFC 4347). DTLS is a relatively recent variant of the mature SSL/TLS family of protocols which adds the capability to secure communication over a connectionless network-layer transport such as UDP. UDP is the preferred network layer transport for the RTPS, as well as for NAT traversal. Like SSL/TLS, the DTLS protocol provides capabilities for certificate-based authentication, data encryption, and message integrity. The protocol specifies a number of standard cryptographic algorithms that must be available; the base set is listed in the TLS 1.1 specification (IETF RFC 4346).

Secure protocol support is provided by the open source OpenSSL library, which has supported the DTLS protocol since the release of OpenSSL 0.9.8. Note however that many critical issues in DTLS were resolved by the OpenSSL 0.9.8f release. For more detailed information about available ciphers, certificate support, etc. please refer to the OpenSSL documentation. The DTLS protocol securely authenticates with each individual peer; as such, multicast communication is not supported by the Secure Transport. There is also a FIPS security-certified version of OpenSSL (OpenSSL-FIPS 1.1.1), but this does not yet support DTLS.

The Secure Transport protocol stack is similar to the Secure WAN transport stack, but without the STUN layer and server. See Figure 4.1 on page 4-3.

4.1.3.1 Security Model

In order to communicate securely, an instance of the secure plugin requires: 1) a certificate authority (shared with all peers), 2) an identifying certificate which has been signed by the authority, 3) the private key associated with the public key contained in the certificate.

The Certificate Authority (CA) is specified by using a PEM format file containing its public key or by using a directory of PEM files following standard OpenSSL naming conventions. If a single CA file is used, it may contain multiple CA keys. In order to successfully communicate with a peer, the CA keys that are supplied must include the CA that has signed that peer's identifying certificate.

The identifying certificate is specified by using a PEM format file containing the chain of CAs used to authenticate the certificate. The identifying certificate must be signed by a CA. It will either be directly signed by a root CA (one of the CAs supplied above), by an authority whose certificate has been signed by the root CA, or by a longer chain of certificate authorities. The file must be sorted starting with the certificate to the highest level (root CA). If the certificate is directly signed by a root CA, then this file will only contain the root CA certificate followed by the identity certificate.

Finally, a private key is required. In order to avoid impersonation of an identity, this should be kept private. It can be stored in its own PEM file specified in one of the private key properties, or it can be appended to the certificate chain file.

One complication in the use of DTLS for communication by *Message Service* is that even though DTLS is a connectionless protocol, it still has client/server semantics. The RTI Secure Transport maps a bidirectional communication channel between two peer applications into a pair of unidirectional encrypted channels. Both peers are playing the part of a client (when sending data) and a server (when receiving).

4.1.3.2 Liveliness Mechanism

When a peer shuts down cleanly, the DTLS protocol ensures that resources are released. If a peer crashes or otherwise stops responding, a liveliness mechanism in the DTLS transport cleans up resources. You can configure the DTLS handshake retransmission interval and the connection liveliness interval.

4.1.4 Certificate Support

Cryptographic certificates are required to use the security features of the WAN transport. This section describes a mechanism to use the OpenSSL command line tool to generate a simple private certificate authority. For more information, see the manual page for the **openssl** tool (http://www.openssl.org/docs/apps/openssl.html) or the book, *"Network Security with OpenSSL"* by Viega, Messier, & Chandra (O'Reilly 2002; http://www.opensslbook.com), or other references on Public Key Infrastructure.

1. Initialize the Certificate Authority:
   a. Create a copy of the openssl.cnf file and edit fields to specify the proper default names and paths.
   b. Create the required CA directory structure:

```
mkdir myCA
mkdir myCA/certs
mkdir myCA/private
mkdir myCA/newcerts
mkdir myCA/crl
touch myCA/index.txt
```

   c. Create a self-signed certificate and CA private key:

```
openssl req -nodes -x509 -days 1095 -newkey rsa:2048 \
   -keyout myCA/private/cakey.pem -out myCA/cacert.pem \
   -config openssl.cnf
```

2. For each identifying certificate:
   a. You may want to create a copy of your customized openssl.cnf file with default identifying information to be used as a template for certificate request creation; the commands below refer to this file as "template.cnf."
   b. Generate a certificate request and private key:

```
openssl req -nodes -new -newkey rsa:2048 -config template.cnf \
    -keyout peer1key.pem -out peer1req.pem
```

   c. Use the CA to sign the certificate request to generate certificate:

```
openssl ca -create serial -config openssl.cnf -days 365 \
    -in peer1req.pem -out myCA/newcerts/peer1cert.pem
```

   d. Optionally, append the private key to the peer certificate:

```
cat myCA/newcerts/peer1cert.pem peer1key.pem \
$>${private location}/ peer1.pem
```

4.1.5 License Issues

The OpenSSL toolkit stays under a dual license, i.e., both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL, please contact **openssl-core@openssl.org**.

```
/* ====================================================================
 * Copyright (c) 1998-2007 The OpenSSL Project.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in
 *    the documentation and/or other materials provided with the
 *    distribution.
 *
 * 3. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 *    acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *
 * THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
 * EXPRESSED OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
```

```
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE OpenSSL PROJECT OR
 * ITS CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
 * NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
 * LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
 * STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
 * ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
 * OF THE POSSIBILITY OF SUCH DAMAGE.
 * ====================================================================
 *
 * This product includes cryptographic software written by Eric Young
 * (eay@cryptsoft.com).  This product includes software written by Tim
 * Hudson (tjh@cryptsoft.com).
 *
 */

 Original SSLeay License
 -----------------------

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given
 * attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 *    The word 'cryptographic' can be left out if the routines from the
 *    library
 *    being used are not cryptographic related :-).
 * 4. If you include any Windows specific code (or a derivative thereof)
 *    from
 *    the apps directory (application code) you must include an
 *    acknowledgement:
 *    "This product includes software written by Tim Hudson
 *    (tjh@cryptsoft.com)"
 *
```

```
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``AS IS'' AND
* ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
* IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
* PURPOSE
* ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
* CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
* STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*
* The licence and distribution terms for any publicly available
* version or
* derivative of this code cannot be changed.  i.e. this code cannot
* simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/
```

4.2 Configuring Secure WAN Transport

The *Secure WAN Transport* package includes two transports:

- **The WAN Transport** is for use on a WAN and includes security.[1] It must be used with the WAN Server, a separate application that provides additional services needed for *Message Service* applications to communicate with each other over a WAN.
- **The Secure Transport** is an alternate transport that provides security on an untrusted LAN. Use of the RTI WAN Server is not required.

These transports can be configured by setting up predefined strings in the Property QoS Policy of the *Connection*.

- WAN Transport Properties (Section 4.2.1)
- Secure Transport Properties (Section 4.2.2)

Refer to the Configuration and Operation Manual for details on these two approaches.

4.2.1 WAN Transport Properties

Table 4.1, “Properties for WAN Transport,” on page 4-12 lists the properties that you can set for the WAN Transport.

4.2.2 Secure Transport Properties

Table 4.2, “Properties for Secure Transport,” on page 4-17 lists the properties that you can set for the Secure Transport.

1. Security is disabled by default.

Table 4.1 **Properties for WAN Transport**

Property Name (prefix with 'dds.transport.WAN.wan1.')[1]	Property Value Description
dds.transport.load_plugins *(note: this does not take a prefix)*	**Required** Comma-separated strings indicating the prefix names of all plug-ins that will be loaded by *Message Service*. For example: "dds.transport.WAN.wan1". You will use this string as the prefix to the property names. Note: you can load up to 8 plug-ins.
library	**Required** Must set to "**libnddstransportwan.so**" (for UNIX) or "**nddstransportwan.dll**" (for Windows). This library and the dependent OpenSSL libraries need to be in the path during run time for use by *Message Service* (in the LD_LIBRARY_PATH environment variable on UNIX/Solaris systems, in PATH for Windows systems).
create_function	**Required** Must be "NDDS_Transport_WAN_create"
aliases	Used to register the transport plug-in returned by NDDS_Transport_WAN_create() (as specified by <WAN_prefix>.create_function) to the *Connection*. Aliases should be specified as a comma-separated string, with each comma delimiting an alias. If it is not specified, the prefix is used as the default alias for the plug-in.
verbosity	Specifies the verbosity of log messages from the transport. Possible values: -1: silent 0 (default): errors only 1: errors and warnings 2: local status 5 or higher: all messages
parent.parent.address_bit_count	Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, the address_bit_count should be set to 8.
parent.parent.properties_bitmap	A bitmap that defines various properties of the transport to the *Message Service* core. Currently, the only property supported is whether or not the transport plug-in will always loan a buffer when *Message Service* tries to receive a message using the plug-in. This is in support of a zero-copy interface.
parent.parent.gather_send_ buffer_count_max	Specifies the maximum number of buffers that *Message Service* can pass to the send() function of the transport plug-in. The transport plug-in send() API supports a gather-send concept, where the send() call can take several discontiguous buffers, assemble and send them in a single message. This enables *Message Service* to send a message from parts obtained from different sources without first having to copy the parts into a single contiguous buffer. However, most transports that support a gather-send concept have an upper limit on the number of buffers that can be gathered and sent. Setting this value will prevent *Message Service* from trying to gather too many buffers into a send call for the transport plug-in.

Table 4.1 **Properties for WAN Transport**

Property Name (prefix with 'dds.transport.WAN.wan1.')[1]	Property Value Description
parent.parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plug-in. This value must be set before the transport plug-in is registered, so that *Message Service* can properly use the plug-in. If you set this higher than the default, then the *Connection's* Receiver-Pool QoS policy's **buffer_size** should also be changed.
parent.parent.allow_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, allow the use of only these interfaces; otherwise allow the use of all interfaces. A comma-separated string, with each comma delimiting an interface. For example, "127.0.0.1, eth0".
parent.parent.deny_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, deny the use of these interfaces. A comma-separated string, with each comma delimiting an interface. For example, "127.0.0.1, eth0". This "black" list is applied after the allow_interfaces_list and filters out the interfaces that should not be used.
parent.send_socket_buffer_size	Size in bytes of the send buffer of a socket used for sending. On most operating systems, setsockopt() will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to the parent.parent.message_size_max property. The maximum value is operating system-dependent.
parent.recv_socket_buffer_size	Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to the property, parent.parent.message_size_max. The maximum value is operating system-dependent.
parent.unicast_enabled	Allows the transport plug-in to use unicast UDP for sending and receiving. By default, it will be turned on. Also by default, it will use all the allowed network interfaces that it finds up and running when the plug-in is instanced.

Table 4.1 **Properties for WAN Transport**

Property Name (prefix with 'dds.transport.WAN.wan1.')[1]	Property Value Description
parent. ignore_loopback_interface	Prevents the transport plug-in from using the IP loopback interface. Three values are allowed: 0: Enable local traffic via this plug-in. This plug-in will only use and report the IP loopback interface only if there are no other network interfaces (NICs) up on the system. 1: Disable local traffic via this plug-in. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plug-in like Shared Memory to talk instead of the IP loopback. -1: Automatic. Lets *Message Service* decide among the above two choices. If a shared memory transport plug-in is available for local traffic, the effective value is 1 (i.e., disable UPV4 local traffic). Otherwise, the effective value is 0, i.e., use UDPv4 for local traffic also.
parent. ignore_nonrunning_interfaces	Prevents the transport plug-in from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed: 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.
parent.no_zero_copy	Prevents the transport plug-in from doing a zero copy. By default, this plug-in will use the zero copy on OSs that offer it. While this is good for performance, it may sometime tax the OS resources in a manner that cannot be overcome by the application. The best example is if the hardware/device driver lends the buffer to the application itself. If the application does not return the loaned buffers soon enough, the node may error or malfunction. In case you cannot reconfigure the H/W, device driver, or the OS to allow the zero copy feature to work for your application, you may have no choice but to turn off zero copy use. By default this is set to 0, so *Message Service* will use the zero-copy API if offered by the OS.

Table 4.1 **Properties for WAN Transport**

Property Name (prefix with 'dds.transport.WAN.wan1.')[1]	Property Value Description
parent.send_blocking	Controls the blocking behavior of send sockets. **Changing this from the default can cause significant performance problems.** Currently two values are defined: NDDS_TRANSPORT_UDPV4_BLOCKING_ALWAYS: Sockets are blocking (default socket options for Operating System). NDDS_TRANSPORT_UDPV4_BLOCKING_NEVER: Sockets are modified to make them non-blocking. **This is not a supported configuration and may cause significant performance problems.**
parent.transport_priority_mask	Mask for the transport priority field. This is used in conjunction with transport_priority_mapping_low/high to define the mapping from transport priority to the IPv4 TOS field. Defines a contiguous region of bits in the 32-bit transport priority value that is used to generate values for the IPv4 TOS field on an outgoing socket. For example, the value 0x0000ff00 causes bits 9-16 (8 bits) to be used in the mapping. The value will be scaled from the mask range (0x0000 - 0xff00 in this case) to the range specified by low and high. If the mask is set to zero, then the transport will not set IPv4 TOS for send sockets.
parent. transport_priority_mapping_low	Sets the low and high values of the output range to IPv4 TOS. These values are used in conjunction with transport_priority_mask to define the mapping from transport priority to the IPv4 TOS field. Defines the low and high values of the output range for scaling. Note that IPv4 TOS is generally an 8-bit value.
parent .transport_priority_mapping_high	
enable_security	Required if you want to use security.
recv_decode_buffer_size	Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data.
port_offset	Port offset to allow coexistence with non-secure UDP transport.
dtls_handshake_resend_interval	DTLS handshake retransmission interval in milliseconds
tls.verify.ca_file	A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for SSL_load_verify_locations for more information. If you want to use security, ca_file or ca_path must be specified; both may be specified.
tls.verify.ca_path	A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format, and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for SSL_CTX_load_verify_locations for more information. If you want to use security, ca_file or ca_path must be specified; both may be specified.
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify.verify_peer	If non-zero, use mutual authentication when performing TLS handshake; if zero, only clients will verify server certificate.

Table 4.1 **Properties for WAN Transport**

Property Name (prefix with 'dds.transport.WAN.wan1.')[1]	Property Value Description
tls.verify.verify_callback	This can be set to one of three values: -- "default" selects the default callback NDDS_Transport_TLS_default_verify_callback() -- "verbose" selects the verbose callback NDDS_Transport_TLS_verbose_verify_callback() -- "none" requests no callback be registered
tls.cipher.cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string.
tls.cipher.dh_param_files	List of available DH key files. For example: "foo.h:512,bar.h:256" means: dh_param_files[0].file = foo.h, dh_param_files[0].bits = 512, dh_param_files[1].file = bar.h, dh_param_files[1].bits = 256,
tls.cipher.engine_id	String ID of OpenSSL cipher engine to request.
tls.identity.certificate_chain_file	**Required if you want to use security.** A string that specifies the name of a file containing an identifying certificate chain (in PEM format). An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If no private key is specified, this file will be used to load a non-RSA private key.
tls.identity.private_key_password	A string that specifies the password for private key.
tls.identity.private_key_file	A string that specifies that name of a file containing private key (in PEM format). If no private key is specified (all values are NULL), this value will default to the same file as the specified certificate chain file.
tls.identity.rsa_private_key_file	A string that specifies that name of a file containing an RSA private key (in PEM format).
transport_instance_id[0] to [NDDS_TRANSPORT_WAN_TRANSPORT_INSTANCE_ID_LENGTH]	**Required** A set of comma-separated values to specify the elements of the array. If less than the full array is specified, it will be right-aligned. For example, the string "01,02" results in the array being set to: {0,0,0,0,0,0,0,0,0,0,1,2}
interface_address	Locator, as a string
server	**Required** Server locator, as a string.
server_port	Server port number.
stun_retransmission_interval	STUN request messages requiring a response are resent with this interval. The interval is doubled after each retransmission. Specified in msec.

Table 4.1 **Properties for WAN Transport**

Property Name (prefix with 'dds.transport.WAN.wan1.')[1]	Property Value Description
stun_number_of_retransmissions	Maximum number of times STUN messages are resent unless a response is received.
stun_liveliness_period	Period at which messages are sent to peers to keep NAT holes open; and to the WAN server to refresh bound ports. Specified in msec.

1. Assuming you used '**dds.transport.WAN.wan1**' as the alias to load the plug-in. If not, change the prefix to match the string used with dds.transport.load_plugins.

Table 4.2 **Properties for Secure Transport**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[1]	Property Value Description
dds.transport.load_plugins *(note: this does not take a prefix)*	**Required** Comma-separated strings indicating the prefix names of all plug-ins that will be loaded by *Message Service*. For example: "dds.transport.DTLS.dtls1". You will use this string as the prefix to the property names. Note: you can load up to 8 plug-ins.
library	**Required** Must set to "libnddstransporttls.so" (for UNIX) or "nddstransporttls.dll" (for Windows). This library and the dependent OpenSSL libraries need to be in the path during run time for use by *Message Service* (in the LD_LIBRARY_PATH environment variable on UNIX systems, in PATH for Windows systems).
create_function	**Required** Must be "NDDS_Transport_DTLS_create"
aliases	Used to register the transport plug-in returned by NDDS_Transport_DTLS_create() (as specified by <DTLS_prefix>.create_function) to the *Connection*. Aliases should be specified as comma separated string, with each comma delimiting an alias. If it is not specified, the prefix (see [1]) is used as the default alias for the plug-in.
network_address	The network address at which to register this transport plug-in. The least significant transport_in.property.address_bit_count will be truncated. The remaining bits are the network address of the transport plug-in. This value overwrites the value returned by the output parameter in NDDS_Transport_create_plugin function as specified in "<DTLS_prefix>.create_function".
verbosity	Specifies the verbosity of log messages from the transport. Possible values: -1: silent 0 (default): errors only 1: errors and warnings 2: local status 5 or higher: all messages

Table 4.2 **Properties for Secure Transport**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[1]	Property Value Description
parent.address_bit_count	Number of bits in a 16-byte address that are used by the transport. Should be between 0 and 128. For example, for an address range of 0-255, the address_bit_count should be set to 8.
parent.properties_bitmap	A bitmap that defines various properties of the transport to the *Message Service* core. Currently, the only property supported is whether or not the transport plug-in will always loan a buffer when *Message Service* tries to receive a message using the plug-in. This is in support of a zero-copy interface.
parent. gather_send_buffer_count_max	Specifies the maximum number of buffers that *Message Service* can pass to the transport plug-in's send() function.
parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plug-in. Note: If you use a value greater than the default, then the Receiver-Pool QoS policy's **buffer_size** on the *Connection* should also be changed.
parent.allow_interfaces	A comma-separated string, with each comma delimiting an interface. For example, "127.0.0.1, eth0".
parent.deny_interfaces	
send_socket_buffer_size	Size in bytes of the send buffer of a socket used for sending.
recv_socket_buffer_size	Size in bytes of the receive buffer of a socket used for sending.
ignore_loopback_interface	Prevents the Transport Plugin from using the IP loopback interface.
ignore_nonrunning_interfaces	Prevents the transport plug-in from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed: 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.
transport_priority_mask	Mask for use of transport priority field.
transport_priority_mapping_low	Low and high values of output range to IPv4 TOS.
transport_priority_mapping_high	
recv_decode_buffer_size	Size of buffer for decoding packets from wire. An extra buffer is required for storage of encrypted data.
port_offset	Port offset to allow coexistence with non-secure UDP transport.
dtls_handshake_resend_interval	DTLS handshake retransmission interval in milliseconds

Table 4.2 **Properties for Secure Transport**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[1]	Property Value Description
dtls_connection_liveliness_ interval	Liveliness interval (multiple of resend interval) The connection will be dropped if no message from the peer is received in this amount of time. This enables cleaning up state for peers that are no longer responding. A secure keep-alive message will be sent every half-interval if no other sends have occurred for a given DTLS connection during that time.
tls.verify.ca_file	A string that specifies the name of file containing Certificate Authority certificates. File should be in PEM format. See the OpenSSL manual page for SSL_load_verify_locations for more information. ca_file or ca_path must be specified; both may be specified.
tls.verify.ca_path	A string that specifies paths to directories containing Certificate Authority certificates. Files should be in PEM format, and follow the OpenSSL-required naming conventions. See the OpenSSL manual page for SSL_CTX_load_verify_locations for more information. ca_file or ca_path must be specified; both may be specified.
tls.verify.verify_depth	Maximum certificate chain length for verification.
tls.verify.verify_peer	If non-zero, use mutual authentication when performing TLS handshake; if zero, only clients will verify server certificate.
tls.verify.verify_callback	This can be set to one of three values: -- "default" selects the default callback NDDS_Transport_TLS_default_verify_callback() -- "verbose" selects the verbose callback NDDS_Transport_TLS_verbose_verify_callback() -- "none" requests no callback be registered
tls.cipher.cipher_list	List of available (D)TLS ciphers. See the OpenSSL manual page for SSL_set_cipher_list for more information on the format of this string.
tls.cipher.dh_param_files	List of available DH key files. For example: "foo.h:512,bar.h:256" means: dh_param_files[0].file = foo.h, dh_param_files[0].bits = 512, dh_param_files[1].file = bar.h, dh_param_files[1].bits = 256,
tls.cipher.engine_id	String ID of OpenSSL cipher engine to request.
tls.identity.certificate_chain_file	**Required** A string that specifies the name of a file containing an identifying certificate chain (in PEM format). An identifying certificate is required for secure communication. The file must be sorted starting with the certificate to the highest level (root CA). If no private key is specified, this file will be used to load a non-RSA private key.
tls.identity.private_key_password	A string that specifies the password for private key.
tls.identity.private_key_file	A string that specifies that name of a file containing private key (in PEM format). If no private key is specified (all values are NULL), this value will default to the same file as the specified certificate chain file.

Table 4.2 **Properties for Secure Transport**

Property Name (prefix with 'dds.transport.DTLS.dtls1')[1]	Property Value Description
tls.identity.rsa_private_key_file	A string that specifies that name of a file containing an RSA private key (in PEM format).

1. Assuming you used 'dds.transport.DTLS.dtls1' as the alias to load the plug-in. If not, change the prefix to match the string used with dds.transport.load_plugins

4.3 Running with RTI Secure WAN Transport

RTI provides optional network transports for communicating across firewalls and for encrypting network packets. These transports constitute the *RTI Secure WAN Transport* product, which must be downloaded and installed separately from *Message Service* itself; see the *RTI Secure WAN Transport Release Notes* and *RTI Secure WAN Transport Installation Guide* for details.

If you do not use *RTI Secure WAN Transport*, you may skip this section. If you do use it, your application will need access to the native libraries described below.

4.3.1 Windows Platforms

To use *Secure WAN Transport*, use the additional libraries from Table 4.3; select the files appropriate for your chosen library format. To use these libraries, add them to your Path environment variable.

Table 4.3 **Additional Libraries for using RTI Secure WAN Transport APIs on Windows Systems**

Description	File Name (Release)	File Name (Debug)
RTI Secure WAN Transport engine[1]	nddstransportwan.dll	nddstransportwand.dll
	nddstransporttls.dll	nddstransporttlsd.dll
OpenSSL[2]	ssleay32.dll	
	libeay32.dll	

1. These libraries are located in <*wan install directory*>\lib\<*architecture*>\, where <*wan install dir*> is where RTI Secure WAN Transport is installed, such as C:\rti\ndds.5.*x.y*.

2. These libraries are located in <*openssl install directory*>\<*architecture*>\lib, where <*openssl install dir*> is where you installed OpenSSL, such as <*openssl install directory*>\<*architecture*>\release\bin.

4.3.2 Linux Platforms

To use **RTI Secure WAN Transport**, use the additional libraries from Table 4.4; select the files appropriate for your chosen library format. To use these libraries, add them to your LD_LIBRARY_PATH environment variable.

Table 4.4 **Additional Libraries for using RTI Secure WAN Transport APIs on Linux Systems**

Description	File Name (Release)	File Name (Debug)
RTI Secure WAN Transport engine[1]	nddstransportwan.so	nddstransportwand.so
	nddstransporttls.so	nddstransporttlsd.so
OpenSSL[2]	libssl.so	
	libcrypto.so	

1. The libraries are located in <*wan install directory*>/lib/<*architecture*>/, where <*wan install directory*> is where you installed RTI Secure WAN Transport, such as /opt/rti/ndds.5.x.y.

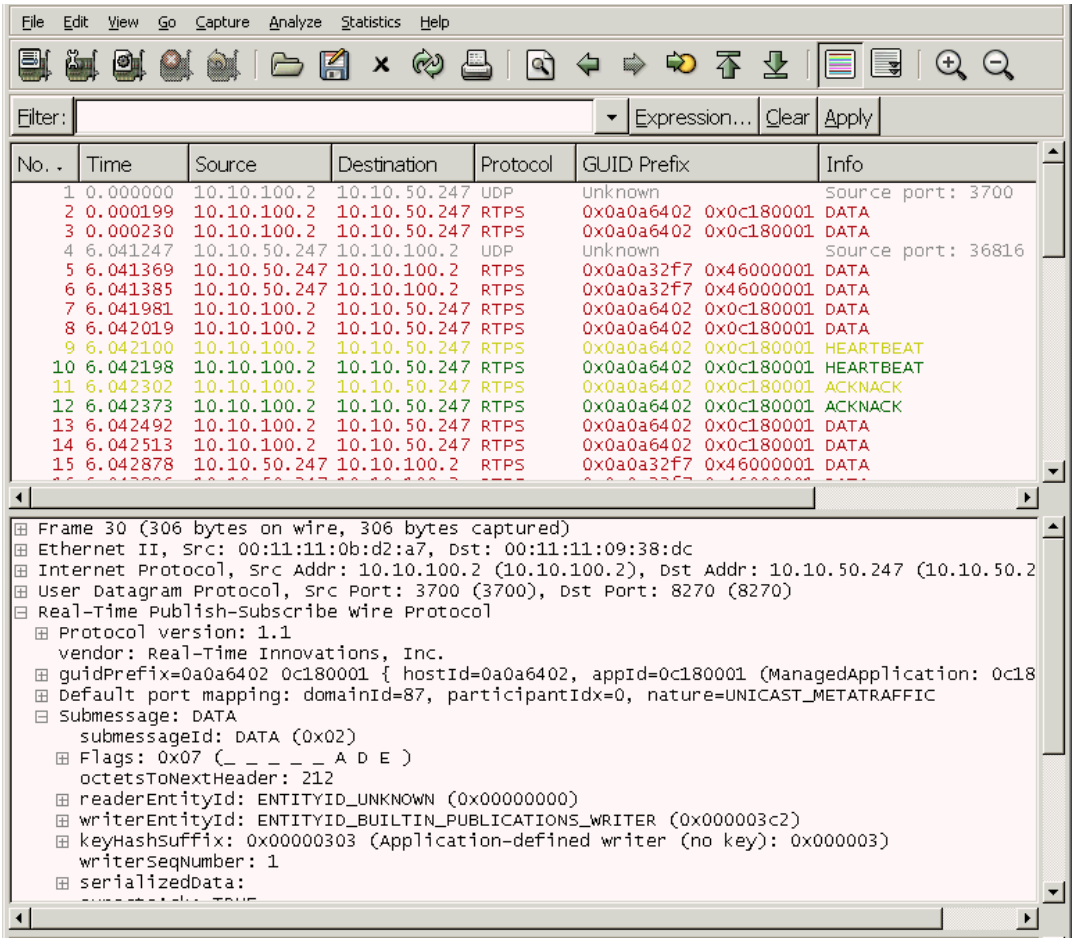
2. These libraries are located <*openssl install directory*>/<*architecture*/lib/, where <*openssl install directory*> is where you installed OpenSSL, such as <*openssl install directory*>/<*architecture*>/release/lib/.

Chapter 5 Interoperability with Wireshark

WireShark is a network packet snooper and dissector. The RTI distribution of this tool contains the latest versions of the RTPS 1.x and 2.x protocol dissectors.

This tool is capable of displaying the contents of network packets originating from a wide variety of products. To view *RTI Message Service* traffic specifically, filter for the "rtps2" protocol. This view will also display packets originating from DDS applications.

Figure 5.1 **Wireshark**



Chapter 6 Application Server Interoperability

Many JMS users also user one or more application servers and use JMS most heavily within the context of managed applications. The JMS and JNDI specifications make it easy to switch out one JMS implementation for another. This chapter describes how to use *RTI Message Service* from an application managed by JBOSS Application Server or Oracle BEA WebLogic Server. Other application servers also allow you to use a "foreign" JMS provider using facilities similar to those described here.

6.1 Using RTI Message Service with JBOSS Application Server

This section describes how to use *RTI Message Service* from an application managed by JBOSS Application Server, version 5.1. It is divided into the following sections:

This section assumes that you have already installed and configured JBOSS according to its *Installation and Getting Started Guide*. The guide for JBOSS Application Server 5.1.0 GA can be found at:
https://www.jboss.org/community/wiki/JBossAS5InstallationandGettingStartedGuide.

6.1.1 Configuring JBOSS Application Server

To use *Message Service* with JBOSS Application Server, you must insert your *Message Service* object configurations into the JBOSS JNDI registry. You will do this with an "external context" declaration. This example uses the *Message Service* object configurations from the "Hello World" example, although you will eventually want to substitute your own configuration file.

Locate the file **${JBOSS_HOME}/server/<*instance-name*>/deploy/messaging/jms-ds.xml**, where:

- **JBOSS_HOME** is your JBOSS installation directory. For example, on Windows it may be **C:\Program Files\jboss-5.1.0.GA.**

- ❏ <*instance-name*> is the name of the JBOSS server configuration you intend to use. If you do not specify such a configuration when you when you start the application server, your configuration is "default." Note that the configuration you choose must be configured for JMS messaging (as the default configuration is).

Add the following bean configuration to this file. When specifying the path to ExampleQosConfig.xml, replace the *x* and *y* in 5.*x*.*y* with the number for your version of *Message Service*:

```
<!-- RTI Message Service configuration: -->
<mbean code="org.jboss.naming.ExternalContext"
       name=":service=ExternalContext,jndiName=rtijms">
   <attribute name="JndiName">rtijms</attribute>
   <attribute name="Properties">
     java.naming.factory.initial=com.rti.jms.JmsConfigContextFactory
     java.naming.provider.url=
file:/C:/Program%20Files/RTI/ndds.5.x.y/example/JMS/ExampleQosConfig.xml
   </attribute>
   <attribute name="InitialContext">
      javax.naming.InitialContext</attribute>
   <attribute name="CacheContext">true</attribute>
   <attribute name="RemoteAccess">false</attribute>
</mbean>
```

The block of XML above defines a JNDI context based on the example object configuration file provided with *Message Service* and adds that context to the JBOSS JNDI registry under the name "rtijms" (the value of the **JndiName** property).

- ❏ You may change the value of the **JndiName** attribute to the name of your choice. Note that this name will appear as a prefix to the name of any *Message Service* object that you look up using JNDI, so if you change it from the value shown here, you will have to make a similar change to the example web application shown below.
- ❏ You must set the value of the **java.naming.provider.url** property to the URL of a valid *Message Service* configuration file. One possible value is given above. Note that you will need to escape any spaces or other characters in the path that are not valid URL characters.

6.1.2 Configuring Your Paths

Message Service consists of two kinds of libraries: Java libraries (Jar files) and native libraries (DLL or SO files, depending on your platform). In order for your managed application to use these libraries, directly or indirectly, WebLogic must have access to them.

- ❏ Java Libraries

  The *Message Service* Jar files need to be added to the class path of the enterprise application that will use the product. See the *Release Notes* for the list of these files and a description of the role of each.

  In a real-world scenario, you may want to package these Jar files into the EAR or WAR files that contain your managed application. However, for the purposes of a first evaluation, you may find it easier to simply copy them into the server's library directory. For example: **${JBOSS_HOME}/server/<*instance-name*>/lib/**.

- ❏ Native Libraries

  Message Service is not implemented in pure Java; it relies on a native-language messaging engine. The *Release Notes* lists the native libraries that provide this implementation. These libraries need to be accessible by the application server; one of the simplest ways to accomplish this is to add them to your system environment.

- On a Windows system, add the directory containing your *Message Service* native libraries to your **Path** environment variable.
- On a UNIX-based system, add the directory containing your *Message Service* native libraries to your **LD_LIBRARY_PATH** environment variable.

6.1.3 Creating and Deploying Your Web Application

Web application hosted by JBOSS Application Server should be packaged as web archive (**.war**) files. For the purposes of this example, we will create the simplest possible web application: a single JSP.

6.1.3.1 Step 1: Create the JSP

In your *Message Service* application:

1. Look up a *ConnectionFactory* using the local JNDI name you defined above.
2. Look up a *Topic* using the local JNDI name you defined above.
3. Write your JMS client code as you otherwise would.
4. For example, paste the following contents into a file **hello.jsp**:

```
<html>
<body>
<p>
Starting...<br/>
<%
    javax.naming.Context ctx = new javax.naming.InitialContext();
    javax.jms.ConnectionFactory cf = (javax.jms.ConnectionFactory)
ctx.lookup("rtijms/hello/helloConnectionFactory");
    javax.jms.Destination topic = (javax.jms.Destination)
ctx.lookup("rtijms/hello/helloTopic");
    javax.jms.Connection conn = cf.createConnection();

    try {
        javax.jms.Session sess = conn.createSession(
false, javax.jms.Session.AUTO_ACKNOWLEDGE);
        javax.jms.Message msg =
sess.createTextMessage("Hello from the JSP");
        javax.jms.MessageProducer pub = sess.createProducer(topic);

        Thread.sleep(2);
        pub.send(msg);
        Thread.sleep(2);

    } finally {
        conn.close();
    }
%><br/>
Message sent!
</p>
</body>
</html>
```

6.1.3.2 Step 2: Create a Web Application Bundle

Every WAR file should contain a file **web.xml**. Create such a file in a directory **WEB-INF** alongside the **hello.jsp** file you created above. Paste the following contents into it:

```
<web-app>
   <display-name>Hello World</display-name>
</web-app>
```

Now that you have JSP and **web.xml** files, bundle them into a WAR file:

```
> ${JAVA_HOME}/bin/jar -cvf helloworld.war hello.jsp WEB-INF
```

You will see a new file **helloworld.war** appear.

6.1.3.3 Step 4: Deploy the WAR File

To deploy your web application, simply copy it into the **deploy** directory of your JBOSS server configuration directory: **${JBOSS_HOME}/server/<*instance-name*>/deploy/**.

While some of the changes you have made can be picked up automatically, others may not be (for example, path modifications). Therefore, if JBOSS is already running, shut it down, close the command window in which it was running, and then restart it now.

Now you're ready to run your web application. By default, it will be available at the URL http://localhost:8080/helloworld/hello.jsp. The example JSP above will generate output like the following:

```
Starting...
Message sent!
```

6.2 Using RTI Message Service with WebLogic Server

This section describes how to use *Message Service* from an application managed by Oracle BEA WebLogic Server, version 9. It is divided into three sections:

- ❏ Configuring WebLogic Server (Section 6.2.1)
- ❏ Configuring Your Paths (Section 6.2.2)
- ❏ Writing Code (Section 6.2.3)

6.2.1 Configuring WebLogic Server

To use *Message Service* with Oracle BEA WebLogic Server, you must define what WebLogic refers to as a foreign JMS server[1]. Only a few steps are required:

6.2.1.1 Step 1: Log into the Administration Console

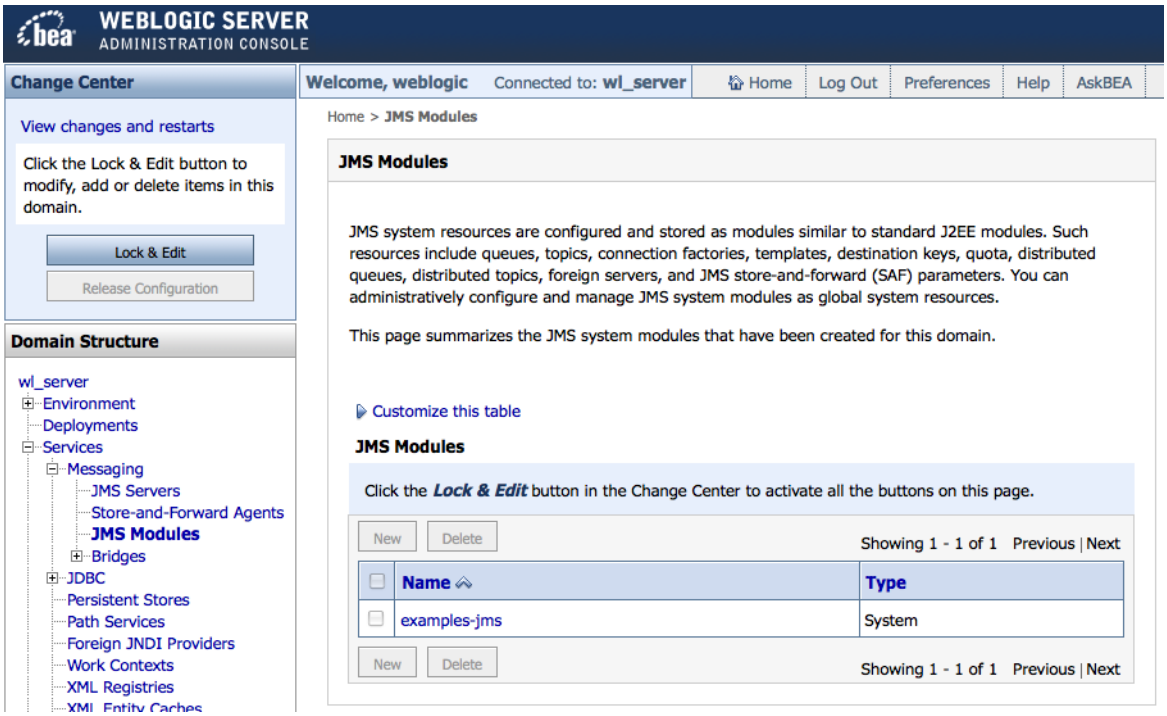
Your WebLogic domain is managed from a web-based application called the WebLogic administration console. If you have just installed WebLogic on your own computer, the console will typically be available at **http://localhost:7001/console**. The default user name and password are both "**weblogic**".

1. The term "server" here is erroneous, since *Message Service* requires no central server.

If you are working with an existing remote installation of WebLogic, talk to your system administrator to learn how to log in.

6.2.1.2 Step 2: Create the Foreign JMS Server

A foreign JMS server belongs to a JMS module. You can find JMS module definitions under **Services, Messaging, JMS Modules**. They are listed in a table on that page.



For a production system, you may want to define a new module. For the purposes of your first test, you will probably find it easier to modify the example JMS module included in your default WebLogic configuration. Click the JMS module to which you want to add the foreign JMS server.

A foreign JMS server is a type of *resource* within the module. To add a new resource, you will need to lock the server configuration for editing. Click the **Lock & Edit** button in the Change Center bar on the left, then click **New** to add a new resource in the table.

- ❏ You will be prompted to select the type of resource to create; select **Foreign Server**.
- ❏ You will also be prompted for a name; choose any name you like.
- ❏ Finally, you will be prompted to specify targeting information. You may accept the defaults or click **Advanced Targeting** to perform further customization.

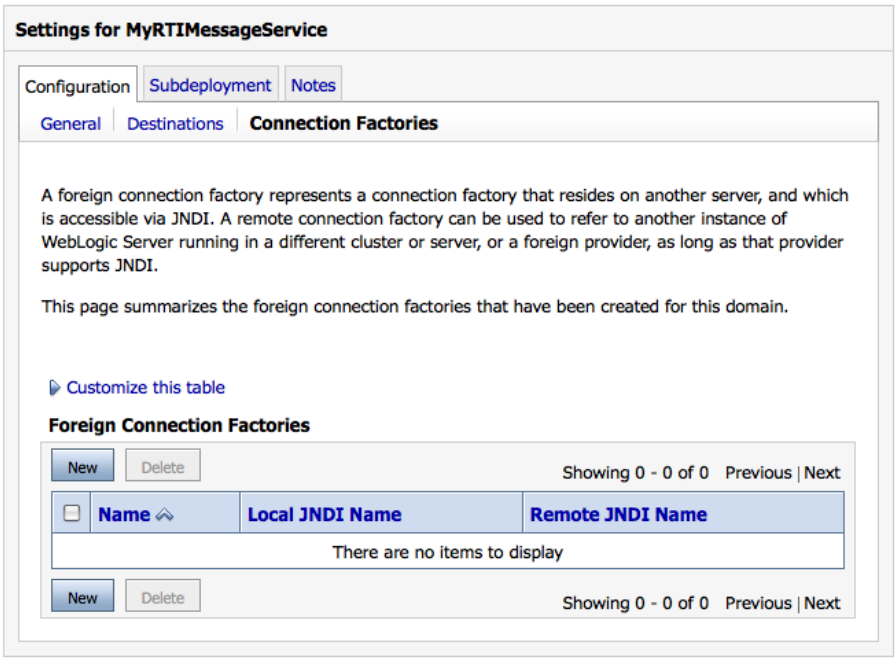
When you click **Finish**, you will be returned to the resource list within the JMS module. Click your new foreign server definition to finish configuring it. You will need to specify two properties on this page:

- ❏ **JNDI Initial Context Factory**: This value must be set to **com.rti.jms.JmsConfigContextFactory**.
- ❏ **JNDI Connection URL**: This value must be a URL pointing to the configuration file you want to use; for example, **file:///C:/RTI/ndds.5.*x.y*/example/JMS/ExampleQosConfig.xml**.

6.2.1.3 Step 3: Create a Foreign ConnectionFactory

For each *ConnectionFactory* you intend to use with *Message Service*, you must configure a JNDI mapping. This mapping will allow the JNDI implementation provided by WebLogic to find the objects in your *Message Service* configuration file.

Select the foreign JMS server you configured earlier, and then navigate to the **Connection Factories** tab. You will see a table labeled **Foreign Connection Factories**; click **New**.



You will be prompted for three strings:

- **Name**: This is a logical, human-readable name of your choice.
- **Local JNDI Name**: This is the name of your *ConnectionFactory* as it will appear in a WebLogic JNDI context.
- **Remote JNDI Name**: This is the name of your *ConnectionFactory* as it appears in your *Message Service* configuration file. When you perform a lookup based on the local JNDI name, the WebLogic JNDI implementation will transparently map that name into this name.

6.2.1.4 Step 4: Create a Foreign Destination

Just as you have to create entries in the WebLogic JNDI repository for all of your *Message Service* connection factories, you also have to create entries for your *Message Service* topics.

Select the foreign JMS server you configured earlier, and then navigate to the tab labeled **Destinations**. You will see a table named **Foreign Destinations**; click **New**. You will be prompted for three strings:

Name: This is a logical, human-readable name of your choice.

Local JNDI Name: This is the name of your *Topic* as it will appear in a WebLogic JNDI context.

Remote JNDI Name: This is the name of your *Topic* as it appears in your *Message Service* configuration file. When you perform a lookup based on the local JNDI name, the WebLogic JNDI implementation will transparently map that name into this name.

6.2.1.5 Step 5: Activate Changes

When you have completed your *ConnectionFactory* and *Topic* configurations, click **Activate Changes** in the Change Center. Your configuration is now active.

6.2.2 Configuring Your Paths

Message Service consists of two kinds of libraries: Java libraries (Jar files) and native libraries (DLL or SO files, depending on your platform). For your managed application to use these libraries, directly or indirectly, WebLogic must have access to them.

❏ Java Libraries

The *Message Service* Jar files need to be added to the class path of the enterprise application that will use the product. See Section 3.2.2 in the RTI Message Service Getting Started Guide for the list of these files and a description of the role of each.

In a real-world scenario, you may want to package these Jar files into the enterprise archive (EAR) files that contain your managed application. However, for the purposes of a first evaluation, you may find it easier to simply copy them into the server's library directory. For example: **$BEA_HOME/weblogic92/samples/domains/wl_server/lib/**.

❏ Native Libraries

Message Service is not implemented in pure Java; it relies on a native-language messaging engine. Section 3.2.3 in the RTI Message Service Getting Started Guide lists the native libraries that provide this implementation. These libraries need to be accessible by the application server; one of the simplest ways to accomplish this is to add them to your system environment.

- On Windows systems, add the directory containing your *Message Service* native libraries to your **Path** environment variable.
- On UNIX-based systems, add the directory containing your *Message Service* native libraries to your **LD_LIBRARY_PATH** environment variable.

6.2.3 Writing Code

Once you have completed the WebLogic configuration and added the *Message Service* libraries to your environment, you are ready to run your code against *Message Service*.

1. Create a JSP, EJB, or other managed application to contain your *Message Service* code.
2. Look up a *ConnectionFactory* using the local JNDI name you defined above.
3. Look up a *Topic* using the local JNDI name you defined above.
4. Write your JMS client code as you otherwise would.

   For example, see the following JSP:

```
<html>
<body>
<%
    javax.naming.Context ctx =          new javax.naming.InitialContext();
           javax.jms.ConnectionFactory cf = (javax.jms.ConnectionFactory)
                                        ctx.lookup("localConnectionFactory-
Name");
```

```
    javax.jms.Destination topic =
                          (javax.jms.Destination) ctx.lookup("localTopic-
Name");

    javax.jms.Connection conn = cf.createConnection();

    try {
           javax.jms.Session sess = conn.createSession(false,
                                              javax.jms.Ses-
sion.DUPS_OK_ACKNOWLEDGE);
           javax.jms.Message msg = sess.createTextMessage(
                                                    "Hello from the
JSP");
           javax.jms.MessageProducer pub = sess.createProducer(topic);
           pub.send(msg);
    } finally {
           conn.close();
    }
%>
Message sent!
</body>
</html>
```

Chapter 7 Complex Event Processing Interoperability

Oracle Complex Event Processing[1] (CEP) is an event server from Oracle Corporation. It provides an application platform for event-driven applications that is based on the OSGi[2] framework, so some of the instructions below may be of interest to users of other OSGi-based products, such as the Eclipse IDE.

The following instructions assume that you have already set up Oracle CEP 10.3 and its Eclipse-based tool chain in your development environment. If you are not able to do so, or if you are using a different version of Oracle CEP, please consult your Oracle product documentation.

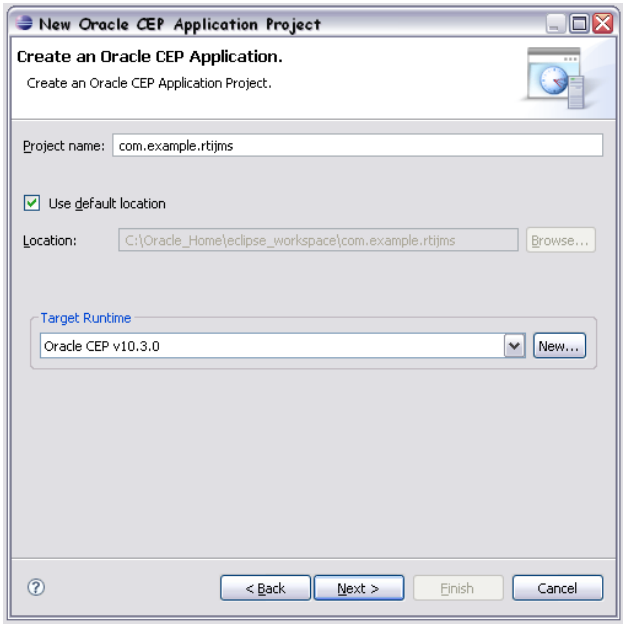
7.1 Create a Project

The Oracle CEP engine, or event server, provides a container (though not a Java EE container) for event-driven applications. Therefore, your first step is to create a project that can be deployed to that container.

This example will not show you how to create an entire event-driven application—please consult your Oracle documentation for that. What it will do is show you how to create simple JMS inbound and outbound adapters and configure them to communicate using *Message Service*.

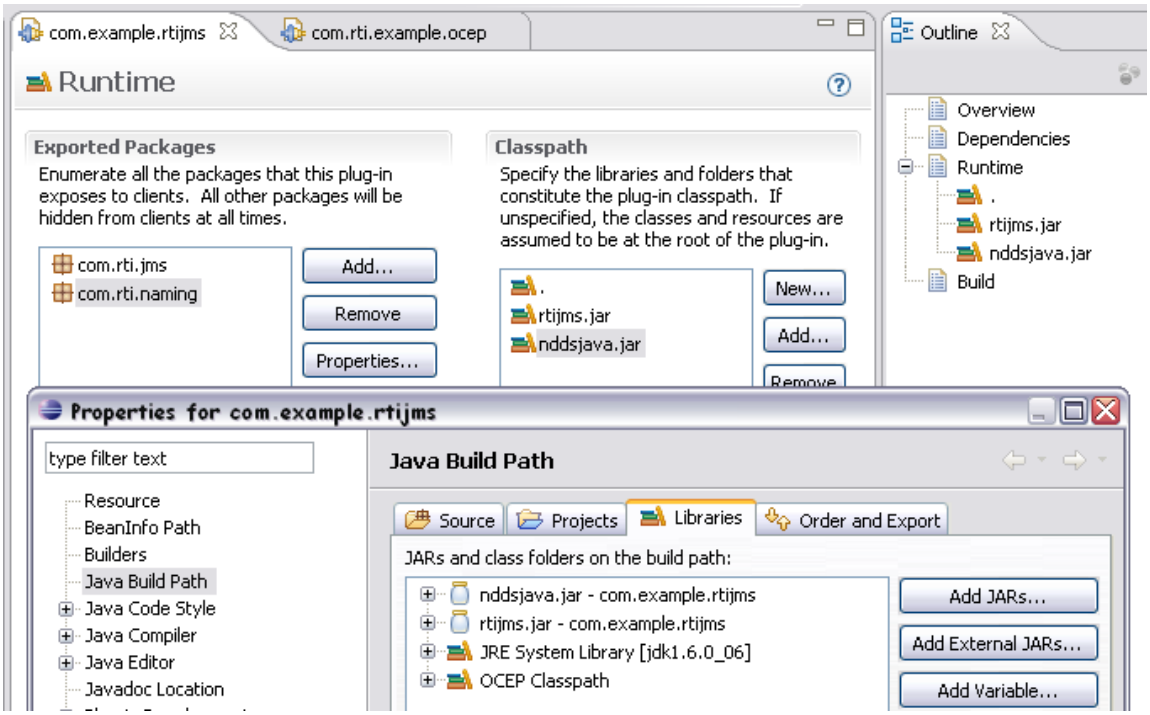
1. http://www.oracle.com/technology/products/event-driven-architecture/index.html
2. http://www.osgi.org/

Open the copy of Eclipse into which you deployed the Java EE and Oracle CEP plug-ins. Create a new project by selecting **File**, **New, Project...** and then selecting **Oracle CEP, Oracle CEP Application Project**. Choose a name for your project. It is not necessary to use a predefined template.



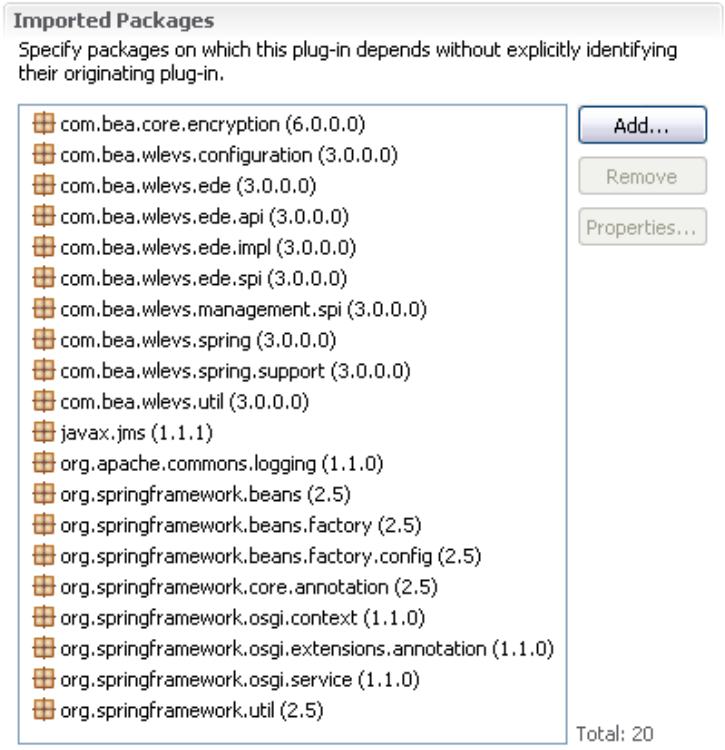
If you have not yet created a target runtime, do that now as well, pointing it to your "BEA Home" (on Windows systems, this directory is **C:\BEA** by default, though you may have chosen another name). In the screenshot above, the chosen target runtime is called **Oracle CEP v10.3.0**.

Once you have created your project, the next step is to add the *Message Service* libraries to it. (If you plan to create multiple applications that all depend on *Message Service*, it may make sense to create a separate OSGi bundle for *Message Service* alone, and then make your project's bundles depend on that shared bundle. For the sake of simplicity, however, these instructions assume that *Message Service* is located within your application's own bundle.)



1. Copy the jar files **rtijms.jar** and **nddsjava.jar** from the **$RTIJMSHOME/class** directory into your project directory.

2. Open your project's manifest file, **META-INF/MANIFEST.MF**, and select the **Runtime** tab. Add these libraries to your project's class path. This step will make these libraries available to your project when it's deployed.
3. **Optional:** If you plan to use any *Message Service*-specific APIs, you will need to add **rti-jms.jar** to the build-time class path, not just the runtime class path. Right-click on your project in the Project Explorer and choose **Properties**. Go to **Java Build Path** and add this jar file in the **Libraries** tab.
4. The JNDI implementation included in *Message Service* requires reflective access to certain packages within the *Message Service* libraries. In order for the OSGi class loaders to locate these packages, they must be exported from your project bundle. Still in the **Runtime** tab of the manifest editor, add **com.rti.jms** and **com.rti.naming** to the list of exported packages.
5. Your project depends on the JMS API; declare that dependency now. Select the **Dependencies** tab of the manifest editor. Under **Imported Packages**, add **javax.jms**. You must also add **com.bea.core.encryption**; this package is required by the Oracle CEP JMS adapters.



Your new project is now ready to use *Message Service*. The next step is to configure the JMS adapters.

7.2 Configure the JMS Adapters

Oracle CEP applications are based on the Spring[1] framework. Configuring the JMS adapters consists of two parts: (1) defining those adapters in a Spring configuration file and then (2) customizing those adapters in the Oracle CEP configuration file.

In this example, we will create one JMS inbound adapter and one JMS outbound adapter, and then route all messages from the inbound adapter directly to the outbound adapter. Consult your Oracle CEP documentation for information about detecting events, transforming data, and other tasks.

7.2.1 Spring Adapter Configuration

The Spring configuration file has already been created for your in your project as **META-INF/spring/<*project name*>.context.xml**. Add the following XML inside the root **beans** element:

```
<wlevs:event-type-repository>
    <wlevs:event-type type-name="SimpleMapEvent">
           <wlevs:metadata>
                  <entry key="message-text" value="java.lang.String" />
           </wlevs:metadata>
    </wlevs:event-type>
</wlevs:event-type-repository>

<wlevs:adapter id="JmsInbound" provider="jms-inbound">
    <wlevs:listener ref="JmsOutbound" />
</wlevs:adapter>

<wlevs:adapter id="JmsOutbound" provider="jms-outbound" />
```

There are three objects being configured here:

1. **Event type:** Events will propagate through your adapters as simple Java map objects. The event type's name is yours to choose. The **entry** element indicates that the event is expected to contain a key by the name **message-text** with a value of type **String**. In your application you can define your message contents however you like.

   By default, Oracle CEP will transform JMS **MapMessages** to and from this map event type. It's possible to define more complex event types, and to define custom mappings to and from other JMS **Message** types, but that configuration is beyond the scope of this document. Please consult your Oracle CEP product documentation.

2. **Inbound JMS adapter:** This adapter identifies an input to your event-driven application. The adapter provider name **jms-inbound** is defined by the built-in Oracle CEP JMS adapter. The adapter ID is yours to choose.

   The **listener** element indicates where events flowing in through this adapter should be routed. In this example, events flow directly to a JMS outbound adapter:

3. **Outbound JMS adapter:** This adapter identifies how events will flow out of your application. The adapter provider name **jms-outbound** is defined by the built-in Oracle CEP JMS adapter. The adapter ID is yours to choose.

7.2.2 Oracle CEP Adapter Configuration

The Oracle CEP adapter configuration can be found in the file **META-INF/wlevs/config.xml**. (The "**wlevs**" acronym that appears in several places refers to the previous name of the Oracle CEP product: WebLogic Event Server.) Delete the existing content inside the root **config** element and replace it with the following:

```
<jms-adapter>
    <name>JmsInbound</name>
    <event-type>SimpleMapEvent</event-type>
```

1. http://www.springsource.org

```
    <jndi-provider-url>
      file:///C:/Oracle_Home/eclipse_workspace/com.example.rtijms/
JmsQosConfig.xml
    </jndi-provider-url>
    <jndi-factory>com.rti.jms.JmsConfigContextFactory</jndi-factory>
    <connection-jndi-name>
      Example JMS Library/DefaultConnectionFactory
    </connection-jndi-name>
    <destination-jndi-name>
      Example JMS Library/InboundTopic
    </destination-jndi-name>
    <concurrent-consumers>1</concurrent-consumers>
    <session-transacted>false</session-transacted>
    <delivery-mode>nonpersistent</delivery-mode>
</jms-adapter>

<jms-adapter>
    <name>JmsOutbound</name>
    <event-type>SimpleMapEvent</event-type>
    <jndi-provider-url>
      file:///C:/Oracle_Home/eclipse_workspace/com.example.rtijms/
JmsQosConfig.xml
    </jndi-provider-url>
    <jndi-factory>com.rti.jms.JmsConfigContextFactory</jndi-factory>
    <connection-jndi-name>
      Example JMS Library/DefaultConnectionFactory
    </connection-jndi-name>
    <destination-jndi-name>
      Example JMS Library/OutboundTopic
    </destination-jndi-name>
    <session-transacted>false</session-transacted>
    <delivery-mode>nonpersistent</delivery-mode>
</jms-adapter>
```

The two adapters configured here correspond to the adapters you defined in the Spring configuration file; the contents of the **name** elements here must be the same as the adapter IDs you defined there. Likewise, the **event-type** here must correspond to an **event-type/type-nam**e element in your Spring configuration.

There are several other parameters to configure. Note that the XML validation that Oracle CEP performs requires these elements to be present and to be in this order; if you omit or reorder the elements, your application may or may not function properly.

- **JNDI provider URL:** The location of your *Message Service* configuration file.
- **JNDI factory:** The fully qualified name of the JNDI factory implementation class that allows your adapters to look up the connection factories and topics defined in the *Message Service* configuration file. This element must take exactly the same value as is shown here.
- **Connection JNDI name, destination JNDI name:** These objects must be defined in your *Message Service* configuration file.
- **Concurrent consumers:** This parameter configures how the JMS adapter uses the JMS API internally. Consult your Oracle CEP documentation for more information about configuring it.
- **Session transacted:** *Message Service* does not support transacted sessions; this value must be false.

- **Delivery mode:** *Message Service* provides an OMG DDS-compatible persistence capability rather than the less flexible JMS-compatible capability. This value must therefore be non-persistent. See the *User's Manual* for more information about configuring *Message Service* for persistent message delivery.

You're done configuring your adapter and ready to run it.

7.3 Run Your Project

Before you can run your application, you must make the *Message Service* native libraries available to that application. There are several ways to do this, including:

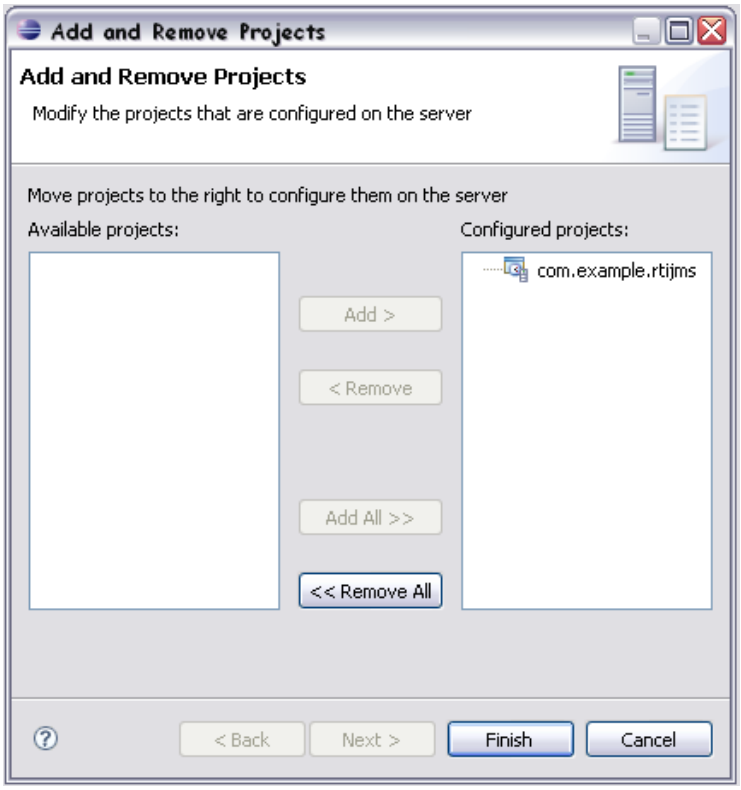
- Place the library directory on your path (the Path environment variable or the **LD_LIBRARY_PATH** variable on UNIX-based systems).
- On Windows systems, copy the libraries into the same directory as your **java** executable.
- Add the libraries to an OSGi bundle and list them in the manifest file with **Bundle-NativeCode**.
- Associate them with your application bundle using a bundle fragment.

Which method you choose is outside the scope of this document. These instructions assume that the native libraries are available in some way; the easiest way to get a simple example up and running is often just to add the library directory to your path.

Running the project is simple. Open the **Servers** view in Eclipse and select the target runtime configuration you created earlier. Then click the **Run** button (a small green arrow) in the upper-right corner of the **Servers** view.

Once the server has started, you need to add your project to the server and start it running. Right-click your server and choose **Add and Remove Projects…**. In the resulting dialog, move

your project from the **Available projects** column to the **Configured projects** column. Then click **Finish**.



Once the project has been added, you will see the server's **State** change to **Started**. The server's **Status** will change to **Republish**. This means that the project has been added to the server but is not yet running. To get the project running, click the **Publish to server** button in the upper right corner of the **Servers** view (the icon looks like a document next to a computer; or, you can right-click on the server and choose **Publish**).

When the server has finished publishing your project, you will see output similar to the following in the **Console** view:

```
<Apr 3, 2011 12:55:06 PM PDT> <Notice> <Server> <BEA-2045000> <The applica-
tion bundle "com.example.rtijms" was deployed successfully to file
[C:\Oracle_Home\user_projects\domains\wlevs30_domain\defaultserver\applica-
tions\com.example.rtijms\com.example.rtijms.jar]>
<Apr 3, 2011 12:55:06 PM PDT> <Notice> <Server> <BEA-2045000> <The applica-
tion bundle "com.example.rtijms" was deployed successfully to file:/C:/
Oracle_Home/user_projects/domains/wlevs30_domain/defaultserver/applica-
tions/com.example.rtijms/com.example.rtijms.jar>
<Apr 3, 2011 12:55:09 PM PDT> <Notice> <Spring> <BEA-2047000> <The applica-
tion context for "com.example.rtijms" was deployed successfully>
```

Congratulations! You have successfully created your first project with Oracle CEP and *Message Service*.