RTI Limited Bandwidth Plugins

User's Manual

Version 7.1.0

Trademarks

Copy and Use Restrictions

Notices

Early Access Software

(4) any such suggestions or ideas you provide regarding the Software (collectively , "Feedback"), may be used and exploited in any and every way by RTI (including without limitation, by granting sub-licenses), on a non-exclusive, perpetual, irrevocable, transferable, and worldwide basis, without any compensation, without any obligation to report on such use, and without any other restriction or obligation to you; and

(5) TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL RTI BE LIABLE TO YOU FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, EXEMPLARY OR PUNITIVE OR CONSEQUENTIAL DAMAGES OF ANY KIND, OR FOR LOST PROFITS, LOST DATA, LOST REPUTATION, OR COST OF COVER, REGARDLESS OF THE FORM OF ACTION WHETHER IN CONTRACT, TORT (INCLUDING WITHOUT LIMITATION, NEGLIGENCE), STRICT PRODUCT LIABILITY OR OTHERWISE, WHETHER ARISING OUT OF OR RELATING TO THE USE OR INABILITY TO USE THE SOFTWARE, EVEN IF RTI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES."

Deprecations and Removals

Any deprecations or removals noted in this document serve as notice under the Real-Time Innovations, Inc. Maintenance Policy #4220 and/or any other agreements by and between RTI and customer regarding maintenance and support of RTI's software.

Deprecated means that the item is still supported in the release, but will be removed in a future release. *Removed* means that the item is discontinued or no longer supported. By specifying that an item is deprecated in a release, RTI hereby provides customer notice that RTI reserves the right after one year from the date of such release and, with or without further notice, to immediately terminate maintenance (including without limitation, providing updates and upgrades) for the item, and no longer support the item, in a future release.

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

The *RTI* ® *Limited Bandwidth Plugins* package includes:

- Limited Bandwidth Participant Discovery Plugin

  Reduces discovery time and network traffic by obtaining some of the information about the participants from an XML file instead of from the normal discovery process, which requires all information to be sent dynamically over the network. All the participants must be known ahead of time and described in an XML file.
- Limited Bandwidth RTPS Transport Plugin

  Reduces the size of the message headers in the Real-Time Publish Subscribe (RTPS) packages sent over the network by the *RTI Connext*® software. The message headers are reduced by eliminating some fields and making other fields smaller.
- Compression Real-Time Publish Subscribe Transport Plugin

  Compresses the RTPS packages sent over the network by *Connext*. You can configure how the packages are compressed, and even provide your own compression algorithm.

You can combine the abilities of these plugins or use them independently. When using more than one of the plugins, their properties must appear in the XML file in the order in which they should be executed.

1.1 Provided Examples

Examples are provided in these directories:

- **<path to examples>/connext_dds/c/limited_bandwidth_plugins**:
  - lbpdiscovery

    - lbrtps
    - zrtps

- **<path to examples>/connext_dds/c++11/limited_bandwidth_plugins**:
    - dil-stacking

1.2 What is a Transport Plugin?

Connext sends data over a variety of transport networks. *Connext* has pluggable transport architecture. The core of *Connext* is transport agnostic, it does not make any assumptions about the actual transports used to send and receive messages.

Connext comes with standard UDPv4/IP and UDPv6/IP pluggable transports, as well as a shared memory transport; these transports are enabled by default. *Connext* also give you the ability to define new transport plugins and run utilizing them.

1.3 What is Discovery?

Discovery is the behind-the-scenes way in which *Connext* objects (*DomainParticipants*, *DataWriters*, and *DataReaders*) find out about each other. Each *DomainParticipant* maintains a database of information about all the active *DataReaders* and *DataWriters* in the same domain. This database is what makes it possible for *DataWriters* and *DataReaders* to communicate. To create and refresh the database, each application follows a common discovery process.

The default discovery mechanism in *Connext* is the one described in the DDS specification and is known as Simple Discovery Protocol, which includes two phases: Simple Participant Discovery and Simple Endpoint Discovery. The goal of these two phases is to build, for each *DomainParticipant*, a complete picture of all the entities that belong to the remote participants in its peers list, which is a list of nodes with which a participant may communicate.

During the Simple Participant Discovery phase, *DomainParticipants* learn about each other. The *DomainParticipant's* details are communicated to all other *DomainParticipants* in the same domain by sending participant declaration messages, also known as participant DATA submessages or participant announcements.

During the Simple Endpoint Discovery phase, *Connext* matches *DataWriters* and *DataReaders*. Information about each application's *DataReaders* and *DataWriters* is exchanged by sending publication/subscription declarations in DATA submessages (participant announcements), which we will refer to as publication DATAs and subscription DATAs. The Simple Endpoint Discovery phase uses reliable communication.

Note: RTI provides two options for endpoint discovery: Simple Endpoint Discovery and Limited Bandwidth Endpoint Discovery. See the Discovery Overview in the RTI Connext Core Libraries User's Manual for more information on these endpoint discovery options.

1.4 Configuring Transports with the Property QoS Policy in XML

Connext provides a mechanism to dynamically load an external transport from an XML QoS profile, like the file generated by rtiddsgen **(USER_QOS_PROFILES.xml)**. The Property QoS policy is used to achieve this purpose.

The Property QoS policy stores name/value (string) pairs that can be used to configure certain parameters of *Connext* that are not exposed through formal QoS policies. *Connext* uses this mechanism to configure external transports.

Syntax for Setting the Property QoS Policy in an XML QoS profile:

```
<qos_library name="Property_Library">
    <qos_profile name="Property_Profile">
        <domain_participant_qos>
            ...
            <property>
                <value>
                    <element>
                        <name>Property1</name>
                        <value>example</value>
                    </element>
                    <element>
                        <name>Property2</name>
                        <value>example</value>
                    </element>
                    ...
                </value>
            </property>
            ...
        </domain_participant_qos>
    </qos_profile>
</qos_library>
```

For more general information, see *Configuring QoS with XML*, in the RTI Connext Core Libraries User's Manual.

For specific information on setting properties for each of the *Limited Bandwidth Plugins*, see their respective chapters in this document.

Chapter 2 Paths Mentioned in Documentation

The documentation refers to:

- **<NDDSHOME>**

  This refers to the installation directory for *RTI® Connext®*. The default installation paths are:

  - macOS® systems:
    /Applications/rti_connext_dds-7.1.0
  - Linux systems, non-*root* user:
    /home/<*your user name*>/rti_connext_dds-7.1.0
  - Linux systems, *root* user:
    /opt/rti_connext_dds-7.1.0
  - Windows® systems, user without Administrator privileges:
    <*your home directory*>\rti_connext_dds-7.1.0
  - Windows systems, user with Administrator privileges:
    C:\Program Files\rti_connext_dds-7.1.0

  You may also see **$NDDSHOME** or **%NDDSHOME%**, which refers to an environment variable set to the installation path.

  Wherever you see **<NDDSHOME>** used in a path, replace it with your installation path.

Note for Windows Users: When using a command prompt to enter a command that includes the path **C:\Program Files** (or any directory name that has a space), enclose the path in quotation marks. For example:

```
"C:\Program Files\rti_connext_dds-7.1.0\bin\rtiddsgen"
```

Or if you have defined the **NDDSHOME** environment variable:

```
"%NDDSHOME%\bin\rtiddsgen"
```

- ***<path to examples>***

  By default, examples are copied into your home directory the first time you run *RTI Launcher* or any script in **<NDDSHOME>/bin**. This document refers to the location of the copied examples as ***<path to examples>***.

  Wherever you see ***<path to examples>***, replace it with the appropriate path.

  Default path to the examples:

  - macOS systems: **/Users/<*your user name*>/rti_workspace/7.1.0/examples**
  - Linux systems: **/home/<*your user name*>/rti_workspace/7.1.0/examples**
  - Windows systems: ***<your Windows documents folder>*\rti_workspace\7.1.0\examples**

    Where 'your Windows documents folder' depends on your version of Windows. For example, on Windows 10, the folder is **C:\Users\<*your user name*>\Documents**.

  Note: You can specify a different location for **rti_workspace**. You can also specify that you do not want the examples copied to the workspace. For details, see *Controlling Location for RTI Workspace and Copying of Examples* in the *RTI Connext Installation Guide*.

Chapter 3 Limited Bandwidth Participant Discovery Plugin

Limited Bandwidth Participant Discovery (LBPD) is achieved with a file-based plugin. Part of the information about the participants is obtained from an XML file instead of being sent dynamically over the network. This method can reduce discovery time and reduce traffic on the network. However, for LBPD to work, all the participants must be known ahead of time and described in an XML file.

The LBPD plugin reduces, but does not eliminate, the network traffic required to exchange participant information. It does this by allowing you to define some of the remote participant data, such as the product version and the RTPS protocol version, in an XML file.

The correlation between the remote participant information defined in the XML file and the information received on the network is done using the RTPS participant identifier (key) first and the participant name second if the identifier is not defined in the XML file (see Table 3.1 Configuration Options for LBPD Plugin for additional details).

This chapter describes how to configure the *RTI Limited Bandwidth Participant Discovery Plugin* and set up your *Connext* application to use the plugin.

You will need two XML files, one for the discovery plugin (see 3.1 Creating the LBPD Plugin Configuration File below) and one for *Connext* (3.2 Configuring the LBPD Plugin in Connext on page 9).

> You must link with the *dynamic* version of the *Connext* libraries. See the RTI Connext Core Libraries Platform Notes for details.

3.1 Creating the LBPD Plugin Configuration File

To use LBPD, you need an XML file that describes all the remote participants. These remote participants must be configured exactly the same as their original QoS properties.

You will specify the name of this file when you configure the plugin in the QoS Profiles XML file (**USER_QOS_PROFILES.xml**) described in 3.2 Configuring the LBPD Plugin in Connext on page 9; see dds.discovery.participant.<string>.config_file

The main structure of this file is:

```
<LBPDiscoveryPluginProfile>
    <participant name="Participant1">
    ...
    </participant>
    <participant name=”Participant2">
    ...
    </participant>
</LBPDiscoveryPluginProfile>
```

Let’s look at an example of this file (you can find it in **<path to examples>/connext_dds/c/limited_bandwidth_plugins/lbpdiscovery/LBPDiscoveryPluginExamplePublisher.xml**).

```
<LBPDiscoveryPluginProfile>
    <participant name="Publisher">
        <key>
            <rtps_host_id>RTPS_AUTO_ID</rtps_host_id>
            <rtps_app_id>RTPS_AUTO_ID</rtps_app_id>
            <rtps_instance_id>RTPS_AUTO_ID</rtps_instance_id>
        </key>
        <rtps_protocol_version>
            <major>2</major>
            <minor>1</minor>
        </rtps_protocol_version>
        <rtps_vendor_id>
            <vendorId>1,1</vendorId>
        </rtps_vendor_id>
        <product_version>
            <major>7</major>
            <minor>x</minor>
            <release>y</release>
            <revision>z</revision>
        </product_version>
        <dds_builtin_endpoints>60</dds_builtin_endpoints>
    </participant>
</LBPDiscoveryPluginProfile>
```

(In <product_version>, *x*, *y* and *z* represent the number of the current release.)

The supported participant configuration options are described in the following tables. They are all optional.

Some descriptions also point out related *Connext* documentation. For example, “See documentation on the Entity QoS policy” means you should see that section in the RTI Connext Core Libraries User's Manual or the *Connext* API Reference HTML documentation**.**

Table 3.1 Configuration Options for LBPD Plugin

Option Name	Option Value and Description

key	The RTPS identifier of the participant. See documentation on the WireProtocol QoS policy. If a key is not set or the values are set to RTPS_AUTO_ID, the correlation between the participant information defined in the XML file and the information received from the network will be done using the participant's entity name.
	Example:
	`<key>`
	`  <rtps_host_id>123</rtps_host_id>`
	`  <rtps_app_id>255</rtps_app_id>`
	`  <rtps_instance_id>348</rtps_instance_id>`
	`</key>`

liveliness_lease_ duration	The liveliness lease duration for the participant.
	Schema:
	`<liveliness_lease_duration>`
	`  <sec>[number | DURATION_ZERO_SEC |`
	`  DURATION_INFINITE_SEC]`
	`  </sec>`
	`  <nanosec>[number | DURATION_ZERO_NSEC |`
	`  DURATION_INFINITE_NSEC]`
	`  </nanosec>`
	`</liveliness_lease_duration>`
	Example:
	`<liveliness_lease_duration>`
	`  <sec>100</sec>`
	`  <nanosec>DURATION_ZERO_NSEC</nanosec>`
	`</liveliness_lease_duration>`

rtps_protocol_ version	The version number of the RTPS protocol being used. See documentation on the ParticipantBuiltinTopicData structure.
	Example:
	`<rtps_protocol_version>`
	`  <major>2</major>`
	`  <minor>1</minor>`
	`</rtps_protocol_version>`

rtps_vendor_id	The identifier of the RTPS vendor. See documentation on ParticipantBuiltinTopicData. RTI's identifier is 1,1. Example: `<rtps_vendor_id><vendorId>1,1</vendorId><rtps_vendor_id>`
participant_name	The name of the remote participant, as set in EntityName QoS policy. Example: `<participant name="Participant1"/>` The participant name is used to correlate the information defined in the XML file with the information received on the network in the absence of the key property. If both the key property and the participant name are not defined, the discovery plugin will report the following error: LBPDiscoveryPluginTypeReaderListenerOnDataAvailable: Cannot find the participant in the database

product_version	The version number for the plugin.
	Example (where *x*, *y*, and *z* represent numbers of the current release):
	`<product_version>`
	`  <major>6</major>`
	`  <minor>x</minor>`
	`  <release>y</release>`
	`  <revision>z</revision>`
	`</product_version>`

dds_builtin_endpoints	Bitmap of builtin endpoints supported by the participant. Each bit indicates a builtin endpoint that may be available on the participant for use in discovery. Example: `<dds_builtin_endpoints>60</dds_builtin_endpoints>`

3.2 Configuring the LBPD Plugin in Connext

This section describes how to configure the properties for the LBPD plugin in the XML QoS Profile file used by *Connext* (such as **USER_QOS_PROFILES.XML**), or in the PropertyQosPolicy for your *Connext* application's *DomainParticipant*. (See the "PROPERTY QosPolicy (DDS Extension)" in the RTI Connext Core Libraries User's Manual.)

Let's look at an example XML file, which you can find in **<path to examples>/connext_dds/c/limited_bandwidth_plugins/lbpdiscovery/USER_QOS_PROFILES.xml**:

```
<domain_participant_qos>
    ...
    <property>
        <value>
            <!-- Specify the library -->
            <element>
                <name>dds.discovery.participant.lbpdiscovery.library</name>
                <value>rtilbpdisc</value>
            </element>
            <!-- Specify the creation function -->
            <element>
                <name>
                    dds.discovery.participant.lbpdiscovery.create_function
                </name>
                <value>DDS_LBPDiscoveryPlugin_create</value>
            </element>
            <!-- Specify the discovery configuration file.
                 Change this property to use your own file. -->
            <element>
                <name>dds.discovery.participant.lbpdiscovery.config_file</name>
                <value>LBPDiscoveryPluginExampleSubscriber.xml</value>
            </element>
            <!-- Load LBP Participant Discovery plugin -->
            <element>
                <name>dds.discovery.participant.load_plugins</name>
                <value>dds.discovery.participant.lbpdiscovery</value>
            </element>
            <!-- Specify the verbosity -->
            <element>
                <name>dds.discovery.participant.lbpdiscovery.verbosity</name>
                <value>0</value>
             </element>
        </value>
    </property>
    ...
</domain_participant_qos>
```

Table 3.2 LBPD Configuration Properties for Connext describes the name/value pairs that you can use to configure the LBPD plugin.

Table 3.2 LBPD Configuration Properties for Connext

Property Name	Property Value and Description

dds.discovery.participant.
load_plugins
	Required.
	String indicating the prefix name of the plugin that will be loaded by *Connext*.
	Set the value to **dds.discovery.participant.<string>**, where <string> can be any string you want, as long as you use the same string consistently for all the properties in this table. Our example uses lbpdiscovery:
	`<element>`
	`<name>dds.discovery.participant.load_plugins</name>`
	`<value>dds.discovery.participant.lbpdiscovery</value>`
	`</element>`

dds.discovery.participant.
<*string*>.library
	Required.
	The name of the dynamic library that contains the LBPD plugin implementation. This library must be in the path during run time for use by *Connext*.
	Set the value to **rtilbpdisc**.
	Example:
	`<element>`
	`<name>`
	`dds.discovery.participant.lbpdiscovery.library`
	`</name>`
	`<value>rtilbpdisc</value>`
	`</element>`

dds.discovery.participant.
<*string*>.create_function
	Required.
	The name of the function that will be called by *Connext* to create an instance of the LBPD plugin.
	Set the value to **DDS_LBPDiscoveryPlugin_create**.
	Example:
	`<element>`
	`<name>`
	`dds.discovery.participant.lbpdiscovery.create_function`
	`</name>`
	`<value>DDS_LBPDiscoveryPlugin_create</value>`
	`</element>`

dds.discovery.participant.
<*string*>.config_file
	Required.
	The name of the discovery configuration file, described in 3.1 Creating the LBPD Plugin Configuration File on page 6.
	Set the value to the name of your own file.
	Example:
	`<element>`
	`<name>`
	`dds.discovery.participant.lbpdiscovery.config_file`
	`</name>`
	`<value>LBPDiscoveryPluginExampleSubscriber.xml</value>`
	`</element>`

Table 3.2 LBPD Configuration Properties for Connext

Property Name	Property Value and Description
dds.discovery.participant.<*string*>.verbosity	Optional. The verbosity for the plugin, for debugging purposes. • -1: Silent • 0: Exceptions only (default) • 1: Warnings • 2 and up: Debug Example: `<element><name>dds.discovery.participant.lbpdiscovery.verbosity</name><value>0</value></element>` Note: the LBPD logging verbosity is per application. The last *DomainParticipant* using LBPD and **explicitly setting this property** will apply that setting to all the *DomainParticipants* using LBPD within the application. If not explicitly set, the verbosity will be left unchanged. Therefore, if no *DomainParticipant* has configured the LBPD verbosity, it will be left to the default value.
dds.discovery.participant.<*string*>.property_validation_action	Optional. By default, property names given in the PropertyQoSPolicy are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the plugin: • VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail. • VALIDATION_ACTION_SKIP: skip validation. • VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail. If this property is not set, the plugin property validation behavior will be the same as that of the *DomainParticipant*, which by default is VALIDATION_ACTION_EXCEPTION. See the "Property Validation" section in the RTI Connext Core Libraries User's Manual.

In addition to setting the properties described above, the **builtin_discovery_plugins** mask (set in the DiscoveryConfigQosPolicy) should be set to SEDP. The default value of this mask is SDP (Simple Discovery Protocol). The SDP consists of two parts, Simple Participant Discovery Protocol (SPDP) and Simple Endpoint Discovery Protocol (SEDP). Using the LBPD plugin replaces the need for the SPDP, so the **builtin_discovery_plugins** should be set to SEDP. This tells *Connext* to only use the SEDP for endpoint discovery, since participant discovery will use the LBPD plugin. If you are using both the LBPD plugin and LBED, this mask should be set to MASK_NONE (for more information on LBED, see "Limited Bandwidth Endpoint Discovery" in the RTI Connext Core Libraries User's Manual).

3.3 Optimizing the Plugin

You can reduce network bandwidth by changing some *Connext* properties in the file **USER_QOS_PROFILE xml**. For an example of this user profile, see the file utils/xml/USER_QOS_PROFILES.xml.

These optimizations apply to the LBPD plugins:

3.3.1 Initial Announcements

When a participant is enabled, by default it sends five announcements. You can reduce the number of initial announcements and the period between them with these properties:

- **initial_participant_announcements**
- **min_initial_participant_announcement_period**
- **max_initial_participant_announcement_period**

Example:

```
<domain_participant_qos>
    <discovery_config>
        <initial_participant_announcements>
            1
        </initial_participant_announcements>
        <min_initial_participant_announcement_period>
            <sec>1</sec>
            <nanosec>0</nanosec>
        </min_initial_participant_announcement_period>
        <max_initial_participant_announcement_period>
            <sec>1</sec>
            <nanosec>0</nanosec>
        </max_initial_participant_announcement_period>
    </discovery_config>
</domain_participant_qos>
```

3.3.2 Liveliness

The participant liveliness period can be increased with the property **participant_liveliness_assert_period**. If this property is increased, the property **participant_liveliness_lease_duration** must also be increased.

Example:

```
<domain_participant_qos>
    <discovery_config>
        <participant_liveliness_lease_duration>
            <sec>1000</sec>
            <nanosec>DURATION_ZERO_NSEC</nanosec>
        </participant_liveliness_lease_duration>
        <participant_liveliness_assert_period>
            <sec>300</sec>
            <nanosec>DURATION_ZERO_NSEC</nanosec>
        </participant_liveliness_assert_period>
    </discovery_config>
```

```
</domain_participant_qos>
```

Chapter 4 Limited Bandwidth RTPS Transport Plugin

The Real-Time Publish Subscribe (RTPS) communication protocol is used by the Data Distribution Service (DDS) interoperability protocol. *Connext* uses RTPS packages to send data over the network. The Limited Bandwidth RTPS (LBRTPS) transport plugin reduces the size of the message headers in the RTPS packages sent over the network. The message headers are reduced by eliminating some fields and making other fields smaller.

This chapter provides a brief overview of how RTPS messages are structured and describes how to configure the LBRTPS transport plugin. More information about RTPS can be found in the OMG Real-Time Publish-Subscribe (RTPS) specification, version 2.5. See also the "Wire Protocol Compatibility" section in the RTI Connext Core Libraries Release Notes for further clarification.

> You must link with the *dynamic* version of the *Connext* libraries. See the RTI Connext Core Libraries Platform Notes for details.

4.1 Understanding the RTPS Message Header

The overall structure of an RTPS *Message* consists of a fixed-size RTPS *Header* followed by a variable number of RTPS *Submessage* parts. Each *Submessage* consists of a *SubmessageHeader* and a variable number of *SubmessageElements*. The RTPS header must appear at the beginning of every message.

The *Header* contains these fields:

- **protocol:** Identifies the message as an RTPS message.

  Representation: 4 bytes

```
0.........8........16........24........32
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

```
|   'R'   |   'T'   |   'P'   |   'S'   |
+---------+---------+---------+---------+
```

- **version:** Identifies the version of the RTPS protocol. .

  Representation: 2 bytes. This release uses version {2, 1}.
- **vendorId:** Indicates the vendor that provides the implementation of the RTPS protocol.

  Representation: 2 bytes. The RTI vendor identifier is {1, 1}.
- **guidPrefix:** Defines a default prefix to use for all GUIDs that appear in the message.

  Representation: 12 bytes: 4 bytes for the host identifier, 4 bytes for the application identifier and 4 bytes for the instance identifier.

Figure 4.1: RTPS Message Structure

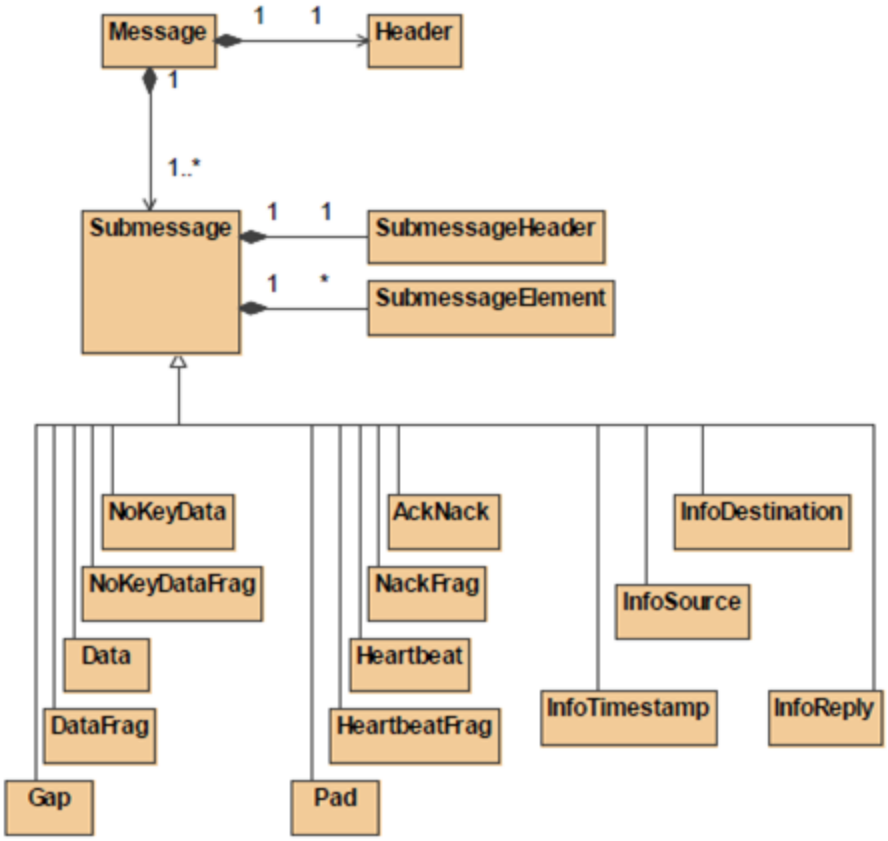
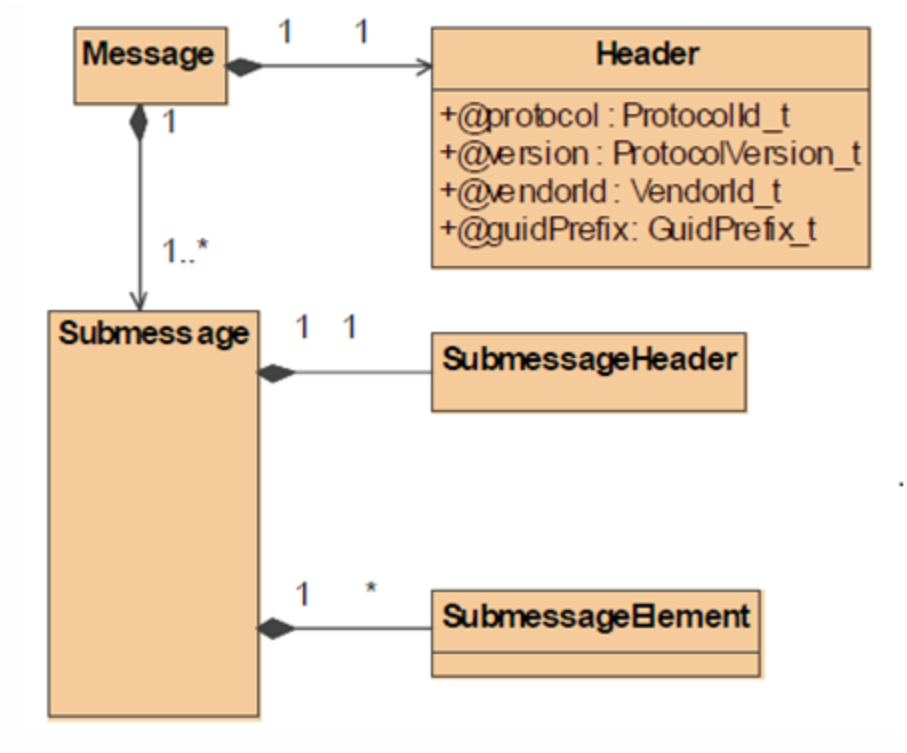


Figure 4.2: RTPS Message Header Structure



The LBRTPS transport reduces the RTPS message headers by eliminating the **protocol**, **version**, and **vendorId** fields in the RTPS *Header* structure.

4.1.1 Submessage Structure

Each RTPS message consists of a variable number of RTPS submessages. All RTPS submessages have the same structure; they start with a *SubmessageHeader,* followed by a concatenation of *SubmessageElement* parts. The *SubmessageHeader* identifies the kind of submessage and the optional elements within that submessage.

The *SubmessageHeader* contains these fields:

- **submessageId:** A 1-byte field that identifies the kind of submessage.
- **flags:** A 1-byte field that identifies the endianness used to encapsulate the submessage, the presence of optional elements within the submessage, and possibly modifies the interpretation of the submessage.
- **submessageLength:** A 2-byte field that indicates the length of the submessage. Given that an RTPS message consists of a concatenation of submessages, the submessage length can be used to skip to the next submessage.

DATA and DATA_FRAG submessages contain a field called **extraflags**. It provides space (2 bytes) for an additional 16 bits of flags beyond the 8 bits provided in the *SubmessageHeader*.

ACKNACK, HEARTBEAT, GAP, ACKNACK_FRAG, HEARTBEAT_FRAG, DATA, and DATA_FRAG submessages contain a reader entity identifier and a writer entity identifier (*readerId* field and *writerId* field). The representation is 4 bytes. These submessages also contain an 8-byte field for a sequence number.

4.2 Configuring the LBRTPS Transport

The LBRTPS transport must be created using the *Connext* Transport API; for more information, please see *Transport Plugins*, in the RTI Connext Core Libraries User's Manual.

Before we describe the name/value pairs that can be used in the Property QoS policy (see the "PROPERTY QosPolicy (DDS Extension)" in the RTI Connext Core Libraries User's Manual) to configure the LBRTPS transport, let's review an example. The first step is to disable the builtin transports by configuring the TransportBuiltin QoS policy with the mask **MASK_NONE**. Then the name/value pairs in the Property QoS policy are set up to load and configure the LBRTPS transport plugin.

```
<qos_library name="Property_Library">
    <qos_profile name="Property_Profile">
        <domain_participant_qos>
        ...
        <transport_builtin>
        <mask>MASK_NONE</mask>
        </transport_builtin>
        <property>
        <value>
                <element>
            <name>dds.transport.load_plugins</name>
            <value>dds.transport.lbrtps</value>
            </element>
            <element>
                <name>dds.transport.lbrtps.library</name>
            <value>rtilbrtps</value>
            </element>
            <element>
            <name>dds.transport.lbrtps.create_function</name>
                <value>LBRTPS_Transport_create_plugin</value>
            </element>
            <element>
                <name>dds.transport.lbrtps.subtransport</name>
            <value>UDPv4</value>
            </element>
            <element>
            <name>dds.transport.lbrtps.aliases</name>
            <value>lbrtps.udpv4</value>
            </element>
            <element>
                <name>
```

```
                        dds.transport.lbrtps.UDPv4.multicast_enabled
                    </name>
            <value>1</value>
            </element>
            <element>
            <name>
                    dds.transport.lbrtps.rtps_header.eliminate_protocol
            </name>
            <value>true</value>
            </element>
            <element>
            <name>
                        dds.transport.lbrtps.rtps_header.eliminate_version
            </name>
            <value>true</value>
            </element>
            <element>
                <name>
                        dds.transport.lbrtps.rtps_header.eliminate_vendorId
                </name>
            <value>true</value>
            </element>
            ...
        </value>
        </property>
        ...
    </domain_participant_qos>
    </qos_profile>
</qos_library>
```

When using the above example, *Connext* will load the LBRTPS transport plugin from the library **rtilbrtps.dll** on Windows systems or **rtilbrtps.so** on Linux systems, and call the function **LBRTPS_Transport_create_plugin()**, which will create the LBRTPS transport. The LBRTPS transport is designed to work over the *Connext* UDPv4 transport. The LBRTPS transport is registered in *Connext* with the participant that uses this QoS profile.

> *Connext* does not assign initial peers to the LBRTPS transport plugin. You can set the initial peers with NDDS_DISCOVERY_PEERS, as described in the *Discovery Overview* chapter in the RTI Connext Core Libraries User's Manual. The LBRTPS transport plugin example uses the NDDS_DISCOVERY_PEERS file to set the multicast address lbrtps://239.255.0.1.

Table 4.1 Configuration Properties for LBRTPS Plugin describes the name/value pairs that you can use to configure the LBRTPS transport.

Table 4.1 Configuration Properties for LBRTPS Plugin

Property Name	Property Value and Description

dds.transport.
load_plugins
	Required.
	Comma-separated strings indicating the prefix names of all plugins that will be loaded by *Connext*.
	Must be set to dds.transport.lbrtps.
	Example:
	`<element>`
	`<name>dds.transport.load_plugins</name>`
	`<value>dds.transport.lbrtps</value>`
	`</element>`

dds.transport.lbrtps. subtransport	Required. Name of the plugin to be loaded by the LBRTPS transport. The LBRTPS transport will work over this loaded plugin. The value can be UDPv4, zrtps, or a user-specified string; see 4.2.1 Configuring the LBRTPS Transport Plugin's 'Subtransport' Property on page 22.

dds.transport.lbrtps.
library
	Required.
	The name of the dynamic library that contains the LBRTPS transport plugin implementation. This library must be in the path during run time for use by *Connext*.
	Example:
	`<element>`
	`<name>dds.transport.lbrtps.library</name>`
	`<value>rtilbrtps</value>`
	`</element>`

dds.transport.lbrtps.
create_function
	Required.
	The name of the function that will be called by *Connext* to create an instance of the LBRTPS transport. The function must have the prototype of NDDS_Transport_create_plugin.
	Must be set to LBRTPS_Transport_create_plugin.
	Example:
	`<element>`
	`<name>dds.transport.lbrtps.create_function</name>`
	`<value>LBRTPS_Transport_create_plugin</value>`
	`</element>`

dds.transport.lbrtps.
aliases
	Required.
	Aliases used to register the LBRTPS transport plugin with the *DomainParticipant*. The transport must have been created by the dds.transport.lbrtps.create_function. Aliases should be specified as comma-separated strings, with each comma delimiting an alias.
	An example alias for the LBRTPS transport, working over a *Connext* UDPv4 transport: lbrtps.udpv4.
	Example:
	`<element>`
	`<name>dds.transport.lbrtps.aliases</name>`
	`<value>lbrtps.udpv4</value>`
	`</element>`

The following properties are optional and appear in alphabetical order.	

dds.transport.lbrtps.
rtps_header.
eliminate_protocol
	Whether or not to eliminate the 4-byte protocol field in the RTPS header.
	Must be a boolean value: true or false.
	Example:
	`<element>`
	`<name>`
	`dds.transport.lbrtps.rtps_header.eliminate_protocol`
	`</name>`
	`<value>true</value>`
	`</element>`

Table 4.1 Configuration Properties for LBRTPS Plugin

Property Name	Property Value and Description

dds.transport.lbrtps.
rtps_header.
eliminate_vendorId
	Whether or not to eliminate the 2-byte vendorId field in the RTPS header.
	Must be a boolean value: true or false.
	Example:
	`<element>`
	`<name>`
	`dds.transport.lbrtps.rtps_header.eliminate_vendorId`
	`</name>`
	`<value>true</value>`
	`</element>`

dds.transport.lbrtps.
rtps_header.
eliminate_version
	Whether or not to eliminate the 2-byte version field in the RTPS header.
	Must be a boolean value: true or false.
	Example:
	`<element>`
	`<name>`
	`dds.transport.lbrtps.rtps_header.eliminate_version`
	`</name>`
	`<value>true</value>`
	`</element>`

dds.transport.lbrtps. rtps_header. reduce_guidPrefix	The reduce_guidPrefix field is comprised of 12 bytes that represent 3 different fields of 4 bytes each: hostId, appId and instanceId. See 4.2.2 Configuring the LBRTPS Transport Plugin's 'reduce_guidPrefix' Property on page 25.

dds.transport.lbrtps.
submessage_header.
combine_
submessageId_
with_flags
	The first two fields in the SubmessageHeader do not use all the bits. The first field, submessageId, only uses 5 bits. The second field, flags, only uses 3 bits. By setting this property to true, these two fields can be packed into a single byte.
	Must be set to a boolean value: true or false.
	Example:
	`<element>`
	`<name>`
	`dds.transport.lbrtps.submessage_header.combine_submessageId_with_flags`
	`</name>`
	`<value>true</value>`
	`</element>`

Table 4.1 Configuration Properties for LBRTPS Plugin

Property Name	Property Value and Description

dds.transport.lbrtps.
submessages.
reduce_entitiesId
	Many submessage kinds include two fields: readerId and writerId.
	In the RTPS protocol specification, these fields are mapped to structure:
	`struct { octet[3] entityKey; octet entityKind; } EntityId_t;`
	As you can see, 4 bytes are allocated for each entity: 3 for the entityKey and 1 for the entityKind. However, you may be able to reduce the size of these fields if you know ahead of time how many *DataReaders* and *DataWriters* you will have.
	If you will have no more than 2048 *DataReaders* and 2048 *DataWriters*, you can reduce the size of each of these fields from four to two bytes. And if you will have no more than 8 *DataReaders* and 8 *DataWriters*, you can reduce each field to only one byte. The math involved is explained below.
	Five bits are always needed for the entityKind. If you have no more than 2048 *DataReaders* and 2048 *DataWriters*, their entityKeys can be 0-2047, which will fit in 11 bits. Thus you only need 2 bytes in this case: 5 bits for entityKind + 11 bits for entityKey = 16 bits = 2 bytes.
	Suppose you have no more than 8 *DataReaders* and 8 *DataWriters*. In this case, you still need 5 bits for the entityKind, but only 3 bits to hold entityKeys 0-7. So you would only need 8 bits: 5 bits for entityKind + 3 bits for entityKey = 8 bits = 1 byte.
	This property's value is expressed as 2 comma-separated integers between 8 and 32, to specify the number of bits to use for the readerId and writerId.
	For example, to reduce both readerId and writerId to 12 bits each:
	`<element> <name>dds.transport.lbrtps.submessages.reduce_entitiesId</name> <value>12,12</value> </element>`

dds.transport.lbrtps. submessages. reduce_ sequenceNumber	Some submessage kinds keep track of the number of submessages received by using an 8-byte sequence number field. You can reduce the number of bytes used for the sequence number by using this property. The value must be an integer between 16 and 64 that specifies the desired size of the sequence number, in bits. Example: `<element> <name> dds.transport.lbrtps.submessages.reduce_sequenceNumber </name> <value>32</value> </element>`

Table 4.1 Configuration Properties for LBRTPS Plugin

Property Name	Property Value and Description

dds.transport.lbrtps.verbosity	The verbosity for the plugin, for debugging purposes.
	• -1: Silent
	• 0: Exceptions only (default)
	• 1: Warnings
	• 2 and up: Debug
	
	Example:
	`<element>`
	`<name>dds.transport.lbrtps.verbosity>/name>`
	`<value>0</value>`
	`</element>`
	
	Note: the LBRTPS logging verbosity is per application. The last *DomainParticipant* using LBRTPS and **explicitly setting this property** will apply that setting to all the *DomainParticipants* using LBRTPS within the application. If not explicitly set, the verbosity will be left unchanged. Therefore, if no *DomainParticipant* has configured the LBRTPS verbosity, it will be left to the default value.

dds.transport.lbrtps.property_validation_action	Optional. By default, property names given in the PropertyQoSPolicy are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the plugin: • VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail. • VALIDATION_ACTION_SKIP: skip validation. • VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail. If this property is not set, the plugin property validation behavior will be the same as that of the *DomainParticipant*, which by default is VALIDATION_ACTION_EXCEPTION. See the "Property Validation" section in the RTI Connext Core Libraries User's Manual.

4.2.1 Configuring the LBRTPS Transport Plugin's 'Subtransport' Property

The required property, **dds.transport.lbrtps.subtransport**, specifies the plugin to be loaded by the LBRTPS transport. The value can be **UDPv4**, **zrtps**, or a user-specified value, as described in the following sections. Once you set the value for a subtransport, the names of all the properties for that subtransport should be in the form **dds.transport.lbrtps.<*subtransport*>.<*property*>**.

4.2.1.1 Using UDPv4 as a Subtransport

To load the *Connext* UDPv4 built-in transport, use the value **UDPv4**. If you want the UDPv4 transport to be created with multicast support, also set **dds.transport.lbrtps.UDPv4.multicast_enabled** to 1, as seen in the example below.

To use the UDPv4 transport and enable multicast:

```
<element>
    <name>dds.transport.lbrtps.subtransport</name>
    <value>UDPv4</value>
</element>
<element>
```

```
    <name>dds.transport.lbrtps.UDPv4.multicast_enabled</name>
    <value>1</value>
</element>
```

4.2.1.2 Using ZRTPS as a Subtransport

In Chapter 5 Compression Real-Time Publish Subscribe Transport Plugin on page 27, you will learn about the Compression Real-Time Publish Subscribe Transport Plugin (ZRTPS). You can use the LBRTPS and ZRTPS transport plugins together, as long as you meet the following three requirements. By using this combination of transport plugins, the LBRTPS transport plugin will reduce the RTPS headers, then the ZRTPS transport plugin will compress the RTPS package.

The LBRTPS transport plugin properties must appear in the XML QoS profile *before* those for the ZRTPS transport plugin. (See the example below.)

As mentioned in Chapter 1 Introduction on page 1, the plugins are executed in the order in which they appear in the XML file. So having the LBRTPS properties appear in the file before the ZRTPS properties is important because once the message is compressed by the ZRTPS transport plugin, the header is no longer recognizable by the LBRTPS transport plugin as an RTPS header.

The ZRTPS transport plugin must be configured as a subtransport of the LBRTPS transport plugin using the value **zrtps**, as seen here:

```
<element>
    <name>dds.transport.lbrtps.subtransport</name>
    <value>zrtps</value>
</element>
```

You will also need to set two additional properties**:**

- **dds.transport.lbrtps.zrtps.library**
- **dds.transport.lbrtps.zrtps.create_function**

```
<element>
    <name>dds.transport.lbrtps.zrtps.library</name>
    <value>rtizrtps</value>
</element>
<element>
    <name>dds.transport.lbrtps.zrtps.create_function</name>
    <value>ZRTPS_Transport_create_plugin</value>
</element>
```

The LBRTPS and ZRTPS transport plugins cannot both use UDPv4 as a subtransport because of port conflict issues.

The following example configures the LBRTPS transport plugin with ZRTPS as a subtransport.

```
<!-- LBRTPS -->
<element>
   <name>dds.transport.load_plugins</name>
   <value>dds.transport.lbrtps</value>
```

```
</element>
<element>
   <name>dds.transport.lbrtps.library</name>
   <value>rtilbrtps</value>
</element>
<element>
   <name>dds.transport.lbrtps.create_function</name>
   <value>LBRTPS_Transport_create_plugin</value>
</element>
<element>
   <name>dds.transport.lbrtps.aliases</name>
   <value>lbrtps.zrtps</value>
</element>
<element>
   <name>dds.transport.lbrtps.subtransport</name>
   <value>zrtps</value>
</element>
<element>
   <name>dds.transport.lbrtps.verbosity</name>
   <value>2</value>
</element>
<!-- ZRTPS-->
<element>
   <name>dds.transport.lbrtps.zrtps.library</name>
   <value>rtizrtps</value>
</element>
<element>
   <name>dds.transport.lbrtps.zrtps.create_function</name>
   <value>ZRTPS_Transport_create_plugin</value>
</element>
<element>
   <name>dds.transport.lbrtps.zrtps.subtransport</name>
   <value>UDPv4</value>
</element>
<element>
   <name>dds.transport.lbrtps.zrtps.UDPv4.multicast_enabled</name>
   <value>1</value>
</element>
<element>
   <name>dds.transport.lbrtps.zrtps.compression_library</name>
   <value>AUTOMATIC_COMPRESSION</value>
</element>
<element>
   <name>dds.transport.lbrtps.zrtps.compression_level</name>
   <value>9</value>
</element>
<element>
    <name>dds.transport.lbrtps.zrtps.verbosity</name>
    <value>0</value>
</element>
```

4.2.1.3 Using a User-Specified Subtransport

To load a user-provided transport plugin, provide a value for **dds.transport.lbrtps.subtransport**, and then use that same value like a prefix to define these two additional properties:

- **dds.transport.lbrtps.*prefix*.library**
- **dds.transport.lbrtps.*prefix*.create_function**

For example, to specify the value, **testplugin**:

```
<element>
    <name>dds.transport.lbrtps.subtransport</name>
    <value>testplugin</value>
</element>
<element>
    <name>dds.transport.lbrtps.testplugin.library</name>
    <value>testplugin</value>
</element>
<element>
    <name>dds.transport.lbrtps.testplugin.create_function</name>
    <value>TestPlugin_Transport_create_function</value>
</element>
```

4.2.2 Configuring the LBRTPS Transport Plugin's 'reduce_guidPrefix' Property

The **dds.transport.lbrtps.rtps_header.reduce_guidPrefix** property is comprised of 12 bytes that represent three different 4-byte fields:

- **hostId**: A machine/OS-specific host identifier, unique in the domain. If you know the number of machines in the network ahead of time, the size of the **hostId** field can be reduced. For example, if only two machines are connected in the network, then this field can be reduced to two bits (the identifiers would be '1' and '2', in binary '01' and '11'). The range of host IDs that can be used in the WireProtocolQosPolicy is therefore limited by this value.
- **appId**: A participant-specific identifier, unique within the scope of the **hostId**. If you know the number of participants in the domain ahead of time, the size of the **appId** field can be reduced. For example, if only four participants are in the domain, then this field can be reduced to three bits. The range of **appId**'s that can be used in the WireProtocolQosPolicy is therefore limited by this value.
- **instanceId**: An instance-specific identifier of the *DomainParticipant* that, together with the **appId**, is unique within the scope of the **hostId**. This identifier is increased each time the participant is recreated in the application. Since most applications create only one *DomainParticipant*, this field can usually be eliminated—in which case a value of 1 is assumed.

The **reduce_guidPrefix** value is expressed as three comma-separated integers between 0 and 32. The values specify the number of bits to use for each identifier (**hostId**, **appId** and **instanceId**).

In the example below, the **hostId** will use 5 bits (because in our example network there will be 32 machines), the **appId** field will use 6 bits (because there will be 64 participants in the domain), and the **instanceId** field will be eliminated. So 11 bits will be used. Then the LBRTPS transport will reduce the **guidPrefix** field 11 bits into 2 bytes.

```
<element>
    <name>
    dds.transport.lbrtps.rtps_header.reduce_guidPrefix
    </name>
    <value>5,6,0</value>
</element>
```

Important Notes:

1. An integral number of bytes is sent by the plugin for the GUID prefix. So the plugin will round up to the next byte when the total number of bits for the IDs add up to less than an integral number of bytes. For example, a **reduce_guidPrefix** value of 3,2,0 requires 5 bits, so 1 byte will be sent; a value of 4,4,2 requires 10 bits so 2 bytes will be sent; etc.
2. Setting any of the **reduce_guidPrefix** fields to 0 will behave as expected on the sending side (no bits are used to send that ID) but on the receiving side, the plugin will use/assume an ID value of 1 for any IDs that were not sent. Thus, when setting 0 bits for an ID field in the **reduce_guidPrefix**, you must use the WireProtocolQosPolicy to set the corresponding ID to a value of 1 in the local participant.
3. When using this property to reduce the number of bits used to encode an ID, the actual ID (host, app, instance) should be a value that can be represented by the reduced bits. If a full ID is used, aliasing of two full ID values to the same reduced ID value may occur since a bit mask is used to convert a full ID to the reduced ID. This could lead to two different participants having the same reduced GUID prefix. For example, using '2,0,0' for the **reduce_guidPrefix** property (2 bits to encode the **host_id** and no bits for the app ID or instance ID) and setting the **rtps_host_id** in the WireProtocolQosPolicy to 3 (0011 in binary) for one participant and 7 (0111 binary) for another participant will cause the reduced **rtps_host_id**'s of both participants to be the same value of 3 (0011 binary). In this situation, discovery will not complete.
4. A **reduce_guidPrefix** value of '0,0,0' is not valid. As noted in point 2 above, a **reduce_guidPrefix** of '0,0,0' will be interpreted as a GUID prefix of host ID = 1, app ID = 1 and instance ID = 1. Unfortunately, this would lead to all participants having the same GUID prefix, since a GUID prefix of host ID = 0, app ID = 0 and instance ID = 0 is not allowed. As a side effect, the GUID prefix cannot be reduced to 0 bytes (1-byte minimum used to send the GUID prefix).
5. When setting the WireProtocolQosPolicy of a participant, a value of 0 for an ID is equivalent to setting the value to DDS_RTPS_AUTO_ID. So when using 0 for an ID, DDS will automatically set the value of the ID to some value other than 0.

Chapter 5 Compression Real-Time Publish Subscribe Transport Plugin

Real-Time Publish Subscribe (RTPS) is the communication protocol used by the Data Distribution Service (DDS) interoperability protocol. *Connext* uses RTPS packages to send data over the network.

The Compression Real-Time Publish Subscribe (ZRTPS) transport plugin reduces the size of the RTPS packages sent over the network by *Connext*.

You can configure the ZRTPS transport plugin to use one of the following algorithms to compress all RTPS packages:

- **Zlib**: This compression library is an abstraction of the DEFLATE compression algorithm used in the gzip file compression program. This is free software, distributed under the ZLIB license.
    - **Windows users: zlib1.dll** is included in the **<NDDSHOME>\lib\<*architecture*>** directory (where NDDSHOME is described in Chapter 2 Paths Mentioned in Documentation on page 4 and <*architecture*> is one of the supported architecture strings listed in the *RTI Connext Core Libraries Release Notes*).
    - **Linux users: libzlib1.so** is likely already installed on your system. If you need to get a Zlib library, please see http://zlib.net for information on how to obtain this library for your platform.
- **Bzip2**: This compression library contains the bzip2 compression algorithm. This algorithm is a lossless data compression algorithm. bzip2 compresses most files more effectively than the older LZW and Deflate compression algorithms, but is considerably slower (~12 times vs. Deflate on typical data). This is free software, distributed under the BSD license.

- **External compression library**: You can add your own compression library and configure the ZRTPS transport plugin to use it.

The transport also supports an automatic mode, which will select from the available algorithms the one that results in the smallest compressed package. While this automatic mode assures the best size reduction, it is slower and uses more memory.

The ZRTPS transport plugin can apply different compression algorithms, depending on each RTPS package's size (small, medium, and large—these sizes are also user-configurable).

The ZRTPS transport works over another transport. It cannot send data over a network by itself. It can work over the *Connext* UDPv4 transport, or a custom transport created using the *Connext* Transports API. You can configure *Connext* to use the ZRTPS transport via the QoS profiles XML; see 5.3 Configuring the ZRTPS Transport on the next page.

> You must link with the **dynamic** version of the *Connext* core libraries. See the *RTI Connext Core Libraries Platform Notes* for details.

5.1 Transport-Related Limitations

The following are known transport-interaction limitations when using the ZRTPS transport plugin:

- Using LBRTPS *and* ZRTPS: See 4.2.1.2 Using ZRTPS as a Subtransport on page 23.
- Neither Shared Memory (SHMEM) nor UDPv6 may be used as subtransports.
- The UDPv4 transport may not be used simultaneously as a transport and a ZRTPS subtransport.

5.2 Differences Between ZRTPS and Built-in Compression

Connext has a built-in data compression feature that you can enable for *DataWriters* and *DataReaders* (for more information, see Data Compression in the RTI Connext Core Libraries User's Manual). ZRTPS compresses RTPS packets, whereas *Connext* built-in compression compresses only the user data payload. They can be used at the same time. The compression methods differ in the following ways:

- ZRTPS is an external plugin and needs to be loaded. Built-in compression is part of the Core Libraries.
- ZRTPS is not announced during discovery. With built-in compression, *DataWriters* and *DataReaders* announce their compression QoS settings to other endpoints, to ensure that the algorithms they use are compatible.
- ZRTPS compresses all RTPS Data packages, even if they correspond to repaired samples (i.e., samples that are lost and need to be resent). Built-in compression keeps the samples compressed in the *DataWriter* queue, so repaired samples are not compressed again.

- ZRTPS only supports the UDPv4 transport or a user-defined transport. Built-in compression can be used with any transport.
- ZRTPS supports ZLIB, BZIP2 and user-defined compression plugins. Built-in compression supports LZ4, ZLIB, and BZIP2.

5.3 Configuring the ZRTPS Transport

This section describes how to configure the properties for the ZRTPS Transport plugin in the XML QoS Profile file used by *Connext* (such as **USER_QOS_PROFILES.XML**), or in the PropertyQosPolicy for your application's *DomainParticipant*. (See the "PROPERTY QosPolicy (DDS Extension)" in the RTI Connext Core Libraries User's Manual.)

Before we describe the name/value pairs that can be used in the Property QoS policy to configure the ZRTPS transport, let's review an example. The first step is to disable the built-in transports by configuring the TransportBuiltin QoS policy with the mask **MASK_NONE**. Then the name/value pairs in the Property QoS policy are set up to load and configure the ZRTPS transport plugin**.**

```
<qos_library name="Property_Library">
    <qos_profile name="Property_Profile">
    <domain_participant_qos>
        ...
            <transport_builtin>
                <mask>MASK_NONE</mask>
            </transport_builtin>
            <property>
                <value>
                <element>
                    <name>dds.transport.load_plugins</name>
                    <value>dds.transport.zrtps</value>
                </element>
                <element>
                    <name>dds.transport.zrtps.library</name>
                    <value>rtizrtps</value>
                </element>
                <element>
                    <name>dds.transport.zrtps.create_function</name>
                    <value>ZRTPS_Transport_create_plugin</value>
                    </element>
                <element>
                    <name>dds.transport.zrtps.subtransport</name>
                    <value>UDPv4</value>
                </element>
                    <element>
                    <name>dds.transport.zrtps.aliases</name>
                        <value>zrtps.udpv4</value>
                </element>
                    <element>
                        <name>dds.transport.zrtps.UDPv4.multicast_enabled</name>
                    <value>1</value>
                </element>
```

```
                <element>
                    <name>dds.transport.zrtps.compression_library</name>
                    <value>AUTOMATIC_COMPRESSION</value>
                </element>
                <element>
                        <name>dds.transport.zrtps.compression_level</name>
                    <value>9</value>
                </element>
                ...
                </value>
            </property>
            ...
        </domain_participant_qos>
    </qos_profile>
</qos_library>
```

When using the above QoS profile, *Connext* will load the ZRTPS transport plugin from the library, **rtizrtps.dll** on Windows systems or **rtizrtps.so** on Linux systems, and call the function **ZRTPS_Transport_create_plugin()** to create the ZRTPS transport.

The ZRTPS transport is designed to work over the *Connext* UDPv4 transport. The automatic mode described previously is set in the ZRTPS transport and the compression level for all compression algorithms is set to 9. The ZRTPS transport will be registered in *Connext* with the participant that uses this QoS profile.

> *Connext* does not assign initial peers to the ZRTPS transport plugin. You can set the initial peers with NDDS_DISCOVERY_PEERS, as described in the chapter on Discovery in the *RTI Connext Core Libraries User's Manual*. The ZRTPS transport plugin example uses the NDDS_DISCOVERY_PEERS file to set the multicast address, zrtps.udpv4://239.255.0.1.

Table 5.1 Configuration Properties for ZRTPS Plugin

Property Name	Property Value and Description

dds.transport.load_plugins	Required.
	Comma-separated strings indicating the prefix names of all plugins to be loaded by *Connext*.
	Set the value to **dds.transport.zrtps**.
	Example:
	`<element>`
	`<name>dds.transport.load_plugins</name>`
	`<value>dds.transport.zrtps</value>`
	`</element>`

dds.transport.zrtps.library	Required.
	The name of the dynamic library that contains the ZRTPS transport plugin implementation. This library must be in the path during run time for use by *Connext*.
	The value must be **rtizrtps**.
	Example:
	`<element>`
	`<name>dds.transport.zrtps.library</name>`
	`<value>rtizrtps</value>`
	`</element>`

Table 5.1 Configuration Properties for ZRTPS Plugin

Property Name	Property Value and Description

dds.transport.zrtps.create_function	Required.
	The name of the function that will be called by *Connext* to create an instance of the ZRPTS transport. The function must have the prototype **NDDS_Transport_create_plugin**.
	Must be set to **ZRTPS_Transport_create_plugin**.
	Example:
	`<element>`
	`  <name>dds.transport.zrtps.create_plugin</name>`
	`  <value>ZRTPS_Transport_create_plugin</value>`
	`</element>`

dds.transport.zrtps.subtransport	Required.
	Name of the plugin to be loaded by the ZRTPS transport. The ZRTPS transport will work over this loaded plugin. The value can be **UDPv4** or a user-specified string; see 5.3.1 Configuring the ZRTPS Transport Plugin's 'Subtransport' Property on page 34.
	Example:
	`<element>`
	`  <name>dds.transport.zrtps.subtransport</name>`
	`  <value>UDPv4</value>`
	`</element>`

dds.transport.zrtps.aliases	Required.
	Aliases used to register the ZRTPS transport plugin with the *DomainParticipant*. The transport must have been created by the dds.transport.zrtps.create_function. Aliases should be specified as a comma-separated string, with each comma delimiting an alias.
	Example:
	`<element>`
	`  <name>dds.transport.zrtps.aliases</name>`
	`  <value>zrtps.udpv4</value>`
	`</element>`

The following properties are optional.	

dds.transport.zrtps.compression_level	Defines the compression level that the compression algorithm will use for all RTPS packages. In automatic mode (see dds.transport.zrtps.compression_library), all compression algorithms will use this level.
	The value must be an integer between 1 and 9 (inclusive). A lower value will result in less time spent doing the compression; a higher value may result in a higher compression percentage (smaller compressed output).
	Example:
	`<element>`
	`  <name>dds.transport.zrtps.compression_level</name>`
	`  <value>9</value>`
	`</element>`

Table 5.1 Configuration Properties for ZRTPS Plugin

Property Name	Property Value and Description

dds.transport.zrtps.compression_level.small_packets
dds.transport.zrtps.compression_level.medium_packets
dds.transport.zrtps.compression_level.large_packets
	Defines the compression level that the compression algorithm will use for small/medium/large RTPS packages. In automatic mode (see dds.transport.zrtps.compression_library), all compression algorithms will use this level.
	The value must be an integer between 1 and 9 (inclusive). A lower value means more speed compression; a higher value means more size reduction.
	Example:
	`<element>`
	`  <name>`
	`  dds.transport.zrtps.compression_level.small_packets`
	`  </name>`
	`  <value>9</value>`
	`</element>`
	`<element>`
	`  <name>`
	`  dds.transport.zrtps.compression_level.medium_packets`
	`  </name>`
	`  <value>9</value>`
	`</element>`
	`<element>`
	`  <name>`
	`  dds.transport.zrtps.compression_level.large_packets`
	`  </name>`
	`  <value>9</value>`
	`</element>`

dds.transport.zrtps.compression_library	Specifies the compression algorithm to be used by the ZRTPS transport for all RTPS packages. These compression algorithms are in external libraries. The ZRTPS transport only uses the libraries whose algorithms will be used. The required libraries must be in the path during run time so the ZRTPS transport can find them.
	There are several compression algorithms that can be used to compress RTPS packages:
	• ZLIB_COMPRESSION: Use the Zlib algorithm from the library **zlib1**.
	• BZIP2_COMPRESSION: Use the Bzip2 algorithm from the library **bzip2**.
	• EXTERNAL_COMPRESSION: Use an external compression library defined in the property dds.transport.zrtps.external_library. See 5.3.2 Configuring the External Compression Library on page 35.
	• AUTOMATIC_COMPRESSION: Compress RTPS packages with all previous compression algorithms and send the smallest package.
	Example:
	`<element>`
	`  <name>`
	`  dds.transport.zrtps.compression_library`
	`  </name>`
	`  <value>AUTOMATIC_COMPRESSION</value>`
	`</element>`

Table 5.1 Configuration Properties for ZRTPS Plugin

Property Name	Property Value and Description

dds.transport.zrtps.compression_library.small_packets
dds.transport.zrtps.compression_library.medium_packets
dds.transport.zrtps.compression_library.large_packets
	Specifies the compression algorithm to be used for small/medium/large RTPS packages. These compression algorithms are in external libraries. The ZRTPS transport only uses the libraries whose algorithms will be used. Required libraries must be in the path during run time so the ZRTPS transport can find them.
	See dds.transport.zrtps.compression_library.
	Example:
	`<element>`
	`<name>`
	`dds.transport.zrtps.compression_library.small_packets`
	`</name>`
	`<value>ZLIB_COMPRESSION</value>`
	`</element>`
	`<element>`
	`<name>`
	`dds.transport.zrtps.compression_library.medium_packets`
	`</name>`
	`<value>ZLIB_COMPRESSION</value>`
	`</element>`
	`<element>`
	`<name>`
	`dds.transport.zrtps.compression_library.large_packets`
	`</name>`
	`<value>ZLIB_COMPRESSION</value>`
	`</element>`

dds.transport.zrtps.external_library	Sets the name of a user's external library that is to be located and used by the ZRTPS transport plugin. See 5.3.2 Configuring the External Compression Library on page 35.
	Example:
	`<element>`
	`<name>dds.transport.zrtps.external_library</name>`
	`<value>external_library.dll</value>`
	`</element>`

dds.transport.zrtps.low_mark,
dds.transport.zrtps.high_mark
	Specifies the size for small, medium, and large RTPS packages. RTPS packages whose size is <= **low_mark** are considered small packages. Package sizes > **low_mark** and <= **high_mark** are considered medium packages. Package sizes > **high_mark** are considered large packages.
	The value for each property is an integer representing a number of bytes.
	Example:
	`<element>`
	`<name>dds.transport.zrtps.low_mark</name>`
	`<value>128</value>`
	`</element>`
	`<element>`
	`<name>dds.transport.zrtps.high_mark</name>`
	`<value>512</value>`
	`</element>`

Table 5.1 Configuration Properties for ZRTPS Plugin

Property Name	Property Value and Description

dds.transport.zrtps.verbosity	The verbosity for the plugin, for debugging purposes.
	• -1: Silent
	• 0: Exceptions only (default)
	• 1: Warnings
	• 2 and up: Debug
	Example:
	`<element>`
	`<name>dds.transport.zrtps.verbosity>/name>`
	`<value>0</value>`
	`</element>`
	Note: the ZRTPS logging verbosity is per application. The last *DomainParticipant* using ZRTPS and **explicitly setting this property** will apply that setting to all the *DomainParticipants* using ZRTPS within the application. If not explicitly set, the verbosity will be left unchanged. Therefore, if no *DomainParticipant* has configured the ZRTPS verbosity, it will be left to the default value.

dds.transport.zrtps.property_validation_action	Optional. By default, property names given in the PropertyQoSPolicy are validated to avoid using incorrect or unknown names (for example, due to a typo). This property configures the validation of the property names associated with the plugin: • VALIDATION_ACTION_EXCEPTION: validate the properties. Upon failure, log errors and fail. • VALIDATION_ACTION_SKIP: skip validation. • VALIDATION_ACTION_WARNING: validate the properties. Upon failure, log warnings and do not fail. If this property is not set, the plugin property validation behavior will be the same as that of the *DomainParticipant*, which by default is VALIDATION_ACTION_EXCEPTION. See the "Property Validation" section in the RTI Connext Core Libraries User's Manual.

5.3.1 Configuring the ZRTPS Transport Plugin's 'Subtransport' Property

The required property, **dds.transport.zrtps.subtransport**, specifies the plugin to be loaded by the ZRTPS transport. Supported subtransports are **UDPv4** or a user-specified transport, as described in the following sections. *Connext* builtin transports other than UDPv4 are not currently supported as subtransports.

> The LBRTPS Transport Plugin cannot be used as a subtransport to ZRTPS. If you want to use both plugins, see 4.2.1.2 Using ZRTPS as a Subtransport on page 23.

5.3.1.1 Using UDPv4 as a Subtransport

To load the *Connext* UDPv4 built-in transport, use the value **UDPv4**. If you want the UDPv4 transport to be created with multicast support, also set **dds.transport.zrtps.UDPv4.multicast_enabled** to 1.

For example:

```
<element>
    <name>dds.transport.zrtps.subtransport</name>
```

```
    <value>UDPv4</value>
</element>
<element>
    <name>dds.transport.zrtps.UDPv4.multicast_enabled</name>
    <value>1</value>
</element>
```

5.3.1.2 Using a User-Specified Subtransport

To load a user-provided transport plugin, provide a value for **dds.transport.zrtps.subtransport**, then use that same value as a prefix to define these two additional properties:

- **dds.transport.zrtps.<*prefix*>.library**
- **dds.transport.zrtps.<*prefix*>.create_function**

For example, to specify the value, **testplugin**:

```
<element>
    <name>dds.transport.zrtps.subtransport</name>
    <value>testplugin</value>
</element>
<element>
    <name>dds.transport.zrtps.testplugin.library</name>
    <value>testplugin</value>
</element>
<element>
    <name>dds.transport.zrtps.testplugin.create_function</name>
    <value>TestPlugin_Transport_create_plugin</value>
</element>
```

5.3.2 Configuring the External Compression Library

The ZRTPS transport plugin loads the external compression library defined in the QoS profile (see dds.transport.zrtps.compression_library). The transport will try to detect the following three functions, which must be implemented in the external library:

- 5.3.2.1 ZRTPS_Transport_external_calculate_length below
- 5.3.2.2 ZRTPS_Transport_external_compress on the next page
- 5.3.2.3 ZRTPS_Transport_external_uncompress on the next page

5.3.2.1 ZRTPS_Transport_external_calculate_length

This function is called before compressing the data.

```
int ZRTPS_Transport_external_calculate_length(int data_length);
```

Parameters:

- **data_length** is the size of the data that will be compressed.

Returns:

- Maximum size of the buffer that the external library needs when the compression function is called.

5.3.2.2 ZRTPS_Transport_external_compress

This function is called when data has to be compressed.

```
int ZRTPS_Transport_external_compress(char *dst,
                      int *dst_length,
                      const char *src,
                      int src_length,
                      int compression_level);
```

Parameters:

- **dst** and **src** are pointers to the destination and source buffers, respectively.
- **dst_length** is the length of the destination buffer. This length is the one returned by **ZRTPS_Transport_external_calculate_length**. After the compression, **dst_length** (which is an input/output variable) must store the actual length of the compressed data.
- **compression_level** is a value between 1 and 9 (inclusive). Some compression libraries use this and others do not.

Returns:

- 0 if successful, -1 if unsuccessful.

5.3.2.3 ZRTPS_Transport_external_uncompress

This function is called to uncompress data.

```
int ZRTPS_Transport_external_uncompress(char *dst,
                          int *dst_length,
                      const char *src,
                    int src_length);
```

Parameters:

- **dst** and **src** are pointers to the destination and source buffers, respectively.
- **dst_length** is the length of the output buffer. After the uncompression, this variable stores the length of the uncompressed data.
- **src_length** is the size of the compressed data in the source buffer.

Returns:

- 0 if successful, -1 if unsuccessful. If successful, **dst_length** must be set to the actual size of the uncompressed data.