Contents

1 Welcome to RTI Queuing Service..........................................................1-1

2 Queuing Service Architecture and Operation ...............................2-1
  2.1 Terms to Know ..................................................................................2-1
  2.2 Paths Mentioned in Documentation ..............................................2-2
  2.3 Load Balancing by Sharing a Queue .............................................2-3
  2.4 DataWriter Connection to a SharedReaderQueue ......................2-5
    2.4.1 QueueProducer Wrapper ......................................................2-5
    2.4.2 Samples with Large Maximum Size ..................................2-5
  2.5 DataReader Connection to a SharedReaderQueue ......................2-6
    2.5.1 QueueConsumer Wrapper ..................................................2-6
    2.5.2 Samples with Large Maximum Size ..................................2-6
  2.6 Queuing Service Entities ..............................................................2-7
  2.7 Sample Distribution to a Selected QueueConsumer ..................2-7
  2.8 Interaction of Publish-Subscribe Entities with Queuing Service Entities........2-8
  2.9 Sample Lifecycle In Queuing Service ...........................................2-9
  2.10 Selecting a QueueConsumer for a Sample ...............................2-11
    2.10.1 Round-Robin Dispatch Policy without Explicit QueueConsumer Availability Feedback 2-11
    2.10.2 Round-Robin Dispatch Policy with Explicit QueueConsumer Availability Feedback 2-12
  2.11 Sending a Reply from QueueConsumer to QueueProducer ..........2-13
    2.11.1 Requester Identification .....................................................2-13
    2.11.2 Request-Reply Correlation ................................................2-14
    2.11.3 Sending the Reply Sample to the Associated Requester .......2-15
    2.11.4 QueueRequester Wrapper .................................................2-15
    2.11.5 QueueReplier Wrapper ....................................................2-15
  2.12 Dead-Letter Queues .................................................................2-15
  2.13 Detecting the Presence of a SharedReaderQueue ......................2-17
  2.14 Queuing Service Persistency .......................................................2-17
    2.14.1 Service State Persistency ..................................................2-17
    2.14.2 SharedReaderQueue Persistency .......................................2-18
      2.14.2.1 The Restore Process ...............................................2-20
  2.15 SharedReaderQueue Resource Management ............................2-20
    2.15.1 Maximum SharedReaderQueue Size .................................2-20
      2.15.1.1 Initial and Maximum Number of Samples ..................2-20
3 Configuring Queuing Service ................................................................. 3-1
  3.1 How to Load the XML Configuration from a File............................ 3-1
  3.2 XML Syntax and Validation ............................................................... 3-3
  3.3 XML Tags for Configuring Queuing Service ..................................... 3-4
    3.3.1 Configuring Queuing Service Types.............................................. 3-4
    3.3.2 Configuring Queuing Service ......................................................... 3-6
    3.3.3 Configuring Queuing Service Administration ................................ 3-7
      3.3.3.1 Configuring Memory Management for a CommandReply Buffer 3-8
    3.3.4 Configuring Queuing Monitoring .................................................. 3-10
      3.3.4.1 Configuring Request-Reply Monitoring .................................... 3-11
      3.3.4.2 Configuring Publish-Subscribe Monitoring ............................... 3-11
        3.3.4.2.1 Publish-Subscribe Monitoring Configuration Inheritance 3-12
      3.3.4.3 Configuring Queuing Service Statistics Calculation Process .... 3-13
        3.3.4.3.1 Statistics Calculation ......................................................... 3-14
    3.3.5 Configuring Queuing Service Persistence Settings ..................... 3-14
    3.3.6 Configuring DomainParticipants .................................................. 3-14
      3.3.6.1 Configuring Memory Management for Sample Buffers ............. 3-18
    3.3.7 Configuring Queuing Service SharedSubscribers ......................... 3-18
    3.3.8 Configuring Session Settings ...................................................... 3-19
    3.3.9 Configuring SharedSubscribers Sessions ....................................... 3-19
    3.3.10 Configuring SharedReaderQueues ................................................. 3-21
    3.3.11 Configuring DeadLetterSharedReaderQueues ................................. 3-26
  3.4 Using Queuing Variables in XML ..................................................... 3-27
  3.5 Enabling RTI Distributed Logger in Queuing Service ......................... 3-27

4 Running Queuing Service ................................................................. 4-1
  4.1 Starting from Launcher ..................................................................... 4-1
  4.2 Starting Manually from the Command Line ....................................... 4-2
  4.3 Using Queuing Service as a Windows Service ................................... 4-4
    4.3.1 Enabling Queuing Service to Run as a Windows Service ............... 4-4
    4.3.2 Running RTI Queuing Service as a Windows Service .................... 4-4
    4.3.3 Notes when Running as a Windows Service .................................. 4-4
    4.3.4 Stopping Queuing Service when it is Running as a Windows Service 4-5
    4.3.5 Disabling Queuing Service when it is Running as a Windows Service 4-5

5 Administering Queuing Service from a Remote Location .......... 5-1
  5.1 Enabling Remote Administration ...................................................... 5-1
  5.2 Remote Administration API ............................................................. 5-1
    5.2.1 Resource Identifiers ................................................................. 5-2
5.2.2 Sample Selector ........................................................................................................... 5-3
5.3 Remote Administration Topics .......................................................................................... 5-4
5.4 Remote Commands in Queuing Service ............................................................................ 5-5
  5.4.1 Create SharedReaderQueue ....................................................................................... 5-5
  5.4.2 Delete SharedReaderQueue ....................................................................................... 5-6
  5.4.3 Flush SharedReaderQueue ......................................................................................... 5-6
  5.4.4 Get SharedReaderQueue Status ................................................................................ 5-6
  5.4.5 Get Service Data ...................................................................................................... 5-7
  5.4.6 Get Samples From a SharedReaderQueue ................................................................. 5-8
  5.4.7 Create SharedSubscriber ....................................................................................... 5-9
  5.4.8 Delete SharedSubscriber ....................................................................................... 5-9
  5.4.9 Shutdown .................................................................................................................. 5-9
5.5 Accessing Queuing Service from a Connext DDS application ........................................ 5-10

6 Publish-Subscribe Monitoring of Queuing Service from a Remote Location .................. 6-1
  6.1 Enabling Publish-Subscribe Monitoring Data .............................................................. 6-1
  6.2 Status Information for a SharedReaderQueue ................................................................ 6-2

7 High Availability .............................................................................................................. 7-1
  7.1 SharedReaderQueue Replication ................................................................................... 7-1
    7.1.1 SharedReaderQueue Replication Protocol ............................................................... 7-1
      7.1.1.1 Sample Replication Phase .................................................................................. 7-2
      7.1.1.2 Enqueue Phase .................................................................................................. 7-3
      7.1.1.3 Consumer Assignment Phase ............................................................................. 7-4
      7.1.1.4 Delivery Phase .................................................................................................. 7-4
    7.1.2 SharedReaderQueue Master Election Protocol ......................................................... 7-4
    7.1.3 SharedReaderQueue Replication Configuration ....................................................... 7-5
      7.1.3.1 Protocol Information Exchange ......................................................................... 7-6
    7.2 Configuration Replication .......................................................................................... 7-7
      7.2.1 SharedReaderQueue for Configuration Replication ................................................ 7-7
    7.3 Replication Clusters .................................................................................................. 7-8

8 Queuing Service Wrapper API ......................................................................................... 8-1
  8.1 QueueProducer Wrapper ............................................................................................. 8-1
  8.2 QueueConsumer Wrapper .......................................................................................... 8-1
  8.3 QueueRequester Wrapper ......................................................................................... 8-1
  8.4 QueueReplier Wrapper ............................................................................................. 8-2

9 Communication Using TCP Transport ............................................................................ 9-1
  9.1 Asymmetric TCP Communication With Queuing Service ........................................... 9-1
  9.2 Asymmetric TCP Communication with Queuing Service And Replication ................. 9-4
Chapter 1  Welcome to RTI Queuing Service

RTI® Queuing Service is a broker that provides a queuing communication model in which a sample is stored in a queue until it is consumed by one QueueConsumer. If there are no QueueConsumers available when the sample is sent, the sample is kept in the queue until a QueueConsumer is available to process it. If a QueueConsumer receives a sample and does not acknowledge it before a specified amount of time or acknowledges it negatively, the sample will be redelivered to a different QueueConsumer.

Queuing Service provides an “at-most-once” and “at-least once” delivery semantic.

By default, Queuing Service keeps the samples in memory. To provide fault tolerance, Queuing Service can be configured to keep the samples on disk.

For high availability, Queuing Service provides mechanisms to replicate its state so that samples can survive the loss of any particular service and/or computer.
Chapter 2  Queuing Service Architecture and Operation

2.1  Terms to Know

You should become familiar with a few key terms and concepts.

- **QueuingService instance**: A single application process (service) that is deployed and configured to host the queues.
- **QueuingServiceName**: A string label that uniquely identifies a QueuingService instance running within a DDS domain.
- **SharedSubscriber**: A container that hosts SharedReaderQueues, allowing remote QueueConsumers to attach to the shared queues, and providing "exactly once" or "at-most once" access to the samples in the shared queues. With these access modes, when one QueueConsumer gets a message, the other QueueConsumers attached to the same SharedReaderQueue do not get that message. A SharedSubscriber can host one or more SharedReaderQueues, each one associated with a different DDS Topic name.
- **SharedSubscriberName**: A string label that uniquely identifies a SharedSubscriber within a DDS domain.
- **SharedReaderQueue**: A logical DataReader queue hosted inside a SharedSubscriber that provides "exactly once" or "at-most once" access to the Consumers attached to the SharedReaderQueue. It is associated with a Topic and the name of the SharedReaderQueue is derived from the name of the Topic and the SharedSubscriber that hosts it. Implementation-wise, a SharedReaderQueue is composed of an input (DDS DataReader) and output (DDS DataWriter) pair that, together with a queue storage, implement the queuing behavior for a Topic. The input DataReader is matched to the DataWriters associated with the Queue Producers and the output DataWriter is matched to the DataReaders associated with the Queue Consumers. The processing logic ensures that each sample in the SharedReaderQueue is delivered to only one of the QueueConsumers.
- **SharedReaderQueueName**: A string label that uniquely identifies a SharedReaderQueue within a DDS domain. It is derived from the name of the SharedSubscriber that hosts the queue and the name of the associated DDS Topic, as in <QueueTopicName>@<SharedSubscriberName>.
- **Session**: Defines a threaded context for a SharedReaderQueue. Sessions are part of SharedSubscribers. SharedReaderQueues in different sessions can be processed in parallel.
- **QueueProducer**: An application-level entity that is either a DDS DataWriter, or a wrapper for it, which allows an application to send data on a single Topic to a SharedReaderQueue.
QueueConsumer: An application-level entity that is either a DDS DataReader, or a simple wrapper for it, which allows an application to access data on a single DDS Topic from a SharedReaderQueue hosted inside a SharedSubscriber. The QueueConsumer DataReaders "compete" for the data on the SharedReaderQueue, such that each sample in the SharedReaderQueue will be received by exactly one QueueConsumer DataReader.

2.2 Paths Mentioned in Documentation

The documentation refers to:

- **<NDDSHOME>**
  
  This refers to the installation directory for Connext® DDS.
  
  The default installation paths are:
  
  - Mac OS X systems:
    
    `/Applications/rti_connext_dds-version`
  
  - UNIX-based systems, non-root user:
    
    `/home/your user name/rti_connext_dds-version`
  
  - UNIX-based systems, root user:
    
    `/opt/rti_connext_dds-version`
  
  - Windows systems, user without Administrator privileges:
    
    `<your home directory>\rti_connext_dds-version`
  
  - Windows systems, user with Administrator privileges:
    
    `C:\Program Files\rti_connext_dds-version` (for 64-bits machines) or `C:\Program Files (x86)\rti_connext_dds-version` (for 32-bit machines)

  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-version\bin\rtiddsgen"

  or if you have defined the NDDSHOME environment variable:

  "%NDDSHOME%\bin\rtiddsgen"

- **RTI Workspace directory, rti_workspace**

  The RTI Workspace is where all configuration files for the applications and example files are located. All configuration files and examples are copied here the first time you run RTI Launcher or any script in <NDDSHOME>/bin. The default path to the RTI Workspace directory is:

  - Mac OS X systems:
    
    `/Users/your user name/rti_workspace`
Load Balancing by Sharing a Queue

- UNIX-based systems:
  `/home/your user name/rti_workspace`
- Windows systems:
  `your Windows documents folder\rti_workspace`

Note: 'your Windows documents folder' depends on your version of Windows. For example, on Windows 7, the folder is `C:\Users\your user name\Documents`; on Windows Server 2003, the folder is `C:\Documents and Settings\your user name\Documents`.

You can specify a different location for the `rti_workspace` directory. See the `RTI Connext DDS Core Libraries Getting Started Guide` for instructions.

- `<path to examples>`

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 these examples as `<path to examples>`. Wherever you see `<path to examples>`, replace it with the appropriate path.

By default, the examples are copied to `rti_workspace/version/examples`

So the paths are:

- Mac OS X systems:
  `/Users/your user name/rti_workspace/version/examples`
- UNIX-based systems:
  `/home/your user name/rti_workspace/version/examples`
- Windows systems:
  `your Windows documents folder\rti_workspace\version\examples`

Note: 'your Windows documents folder' is described above.

You can specify that you do not want the examples copied to the workspace. See the `RTI Connext DDS Core Libraries Getting Started Guide` for instructions.

2.3 Load Balancing by Sharing a Queue

A DDS `DataReader` has an ingress ReaderQueue that stores received samples. The `DataReader` can perform a `take()` operation to remove the data from the ReaderQueue, in which case a subsequent read/take will not see that sample. Two threads can read/take from the same `DataReader` to balance the load of processing samples from the queue. However, each `DataReader` has a different ReaderQueue; therefore, they are independent from each other. "Taking" from one `DataReader` does not affect the other `DataReaders`.

`Queuing Service` provides a way to share a ReaderQueue (SharedReaderQueue) among `DataReaders` of the same `Topic` (see `Figure 2.1`) running in separate processes, possibly on different computers. By sharing the same ReaderQueue, multiple `DataReaders` can collaborate, coordinate, and load-balance among each other.
Realizing the SharedReaderQueue in a separate service also decouples the lifecycle of the samples from that of the producer (DataWriter) and consumer (DataReader) of the data.

In order to be shared, a ReaderQueue must have a ReaderQueueName, so that a DataReader can specify which queue to attach to.

*Queuing Service* provides a way to host the SharedReaderQueues. DataReaders attach to a shared ReaderQueue by specifying the same ReaderQueueName. Multiple DataReaders can attach to the same shared ReaderQueue and *Queuing Service* will ensure that each sample is delivered to exactly one DataReader.

SharedReaderQueues exist within SharedSubscribers. A SharedSubscriber has a name (SharedSubscriberName) that provides a scope for the shared ReaderQueue names. Each SharedReaderQueue is associated with exactly one DDS *Topic*. A single SharedSubscriber is not allowed to host two SharedReaderQueues of the same Topic name; hence the Topic name uniquely identifies the SharedReaderQueue within the SharedSubscriber. For this reason, the name of a shared ReaderQueue is defined by combining the two, as in: *aTopicName@aSharedSubscriberName*. 
2.4 DataWriter Connection to a SharedReaderQueue

You can use a DataWriter to send data to a SharedReaderQueue. The DataWriter simply writes to the Topic that is associated with a SharedReaderQueue.

With regards to QoS, the DataWriter can specify any DataWriter QoS, except: the following are required:

- reliability.kind = RELIABLE_RELIABILITY_QOS
- reliability.acknowledgment_kind = APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE
- history.kind = KEEP_ALL_HISTORY_QOS

The DataWriter is typically also configured with durability.kind set to VOLATILE_DURABILITY_QOS.

For every received sample, Queuing Service sends an application-level acknowledgement (AppAck) message to the QueueProducer's DataWriter indicating successful processing or rejection the sample.

The sending of the application-level acknowledgement (enabled by default) message is optional and can be disabled on a per SharedReaderQueue basis by setting the tag <app_ack_sample_to_producer> under <queue_qos>/<reliability> to false (see Table 3.15, Queue QoS Tags).

Samples are successfully processed when they are stored in the SharedReaderQueue. Samples are rejected when they cannot be stored in the SharedReaderQueue.

One possible cause of rejection is when the maximum number of samples that can be stored in the queue is exceeded.

The response data of the AppAck message for successfully processed samples will be a single byte set to 1. The response data for rejected samples will be a single byte set to 0.

You may want to capture the AppAck message by installing a listener on the DataWriter that implements the on_application_acknowledgment() callback.

2.4.1 QueueProducer Wrapper

To simplify the use and configuration of a DataWriter to send samples to a SharedReaderQueue, Connext DDS provides an abstraction, QueueProducer<aMessageType>, that wraps the DataWriter and provides additional services such as an operation to detect if there is a matching SharedReaderQueue or an operation to wait for application-level acknowledgement after sending a sample.

For more information, see Chapter 8: Queuing Service Wrapper API.

Note: In this release, the QueueProducer wrapper API is only supported in the .NET API.

2.4.2 Samples with Large Maximum Size

By default, Connext DDS preallocates the samples in the QueueProducer’s DataWriter queue and the keys stored with the instances to their maximum size. If the SharedReaderQueue type has variable-size members (sequences and/or strings) with large maximum size this may lead to high memory-usage.

For information on how to reduce memory consumption on a DataWriter, see Sample-Data and Instance-Data Memory Management in the RTI Connext DDS Core Libraries User’s Manual.
2.5 DataReader Connection to a SharedReaderQueue

You can use a DataReader to read samples from a SharedReaderQueue as long as the DataReader is configured as follows:

- The DataReader must attach to the SharedSubscriber that contains the SharedReaderQueue. It does this by setting the property dds.data_reader.shared_subscriber_name in reader_qos.property with a value that is equal to the SharedSubscriberName. This property must be propagated as follows:

  ```xml
  <element>
    <name>dds.data_reader.shared_subscriber_name</name>
    <value>MySharedSubscriberName</value>
    <propagate>true</propagate>
  </element>
  ```

- The DataReader must set a ContentFilteredTopic on the related_reader_guid. Queuing Service uses this filter to distribute a sample only to the DataReader that has been selected for the sample (see Sample Distribution to a Selected QueueConsumer (Section 2.7)).

- The DataReader must subscribe to the Topic `<SharedReaderQueue Topic-Name>@<SharedSubscriberName>`. With regards to QoS, the DataReader can specify any DataReader QoS except: reliability.kind must be set to RELIABLE_RELIABILITY_QOS and reliability.acknowledgment_kind must be set to APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE.

  The DataReader is typically also configured with durability.kind set to VOLATILE_DURABILITY_QOS.

  The application must acknowledge the successful processing or rejection of a received sample using the DataReader’s acknowledge_sample() and/or acknowledge_all() operations.

  The response data for successfully processed samples will be a single byte set to 1. The response data for rejected samples will be a single byte set to 0.

  For more information on the sample lifecycle in a SharedReaderQueue, see Sample Lifecycle In Queuing Service (Section 2.9).

2.5.1 QueueConsumer Wrapper

To simplify the use and configuration of a DataReader to receive samples from a SharedReaderQueue, Connext DDS provides an abstraction, QueueConsumer<MessageType>, which wraps the DataReader and provide additional services such as an operation to detect if there is a matching SharedReaderQueue or a blocking operation to receive samples.

For more information, see Chapter 8: Queuing Service Wrapper API.

Note: In this release, the QueueConsumer wrapper API is only supported in the .NET API.

2.5.2 Samples with Large Maximum Size

By default, Connext DDS preallocates the samples in the QueueConsumer’s DataReader queue and the keys stored with the instances to their maximum size. If the SharedReaderQueue type has variable-size members (sequences and/or strings) with large maximum size, this may lead to high memory-usage.

For information on how to reduce memory consumption on a DataReader, see Sample-Data and Instance-Data Memory Management in the RTI Connext DDS Core Libraries User’s Manual.
2.6 Queuing Service Entities

A SharedReaderQueue is the result of the association of a Topic with a SharedSubscriber. For each SharedReaderQueue, Queuing Service instantiates:

- A DataReader to receive data from the QueueProducer<aMessageType>
- A DataWriter to send data to the QueueConsumer<aMessageType>

In the entities above, aMessageType refers to the data type of the Topic associated with the SharedReaderQueue.

The Queuing Service DataReader subscribes directly to the SharedReaderQueue Topic with name aTopicName. Thus the Queuing Service DataReader which will 'match' the QueueProducer<aMessageType> DataWriter, subject to normal DDS type and QoS matching.

The Queuing Service DataWriter publishes a Topic whose name is obtained by concatenating the SharedReaderQueue Topic name aTopicName with the SharedSubscriber name aSharedSubscriberName as in aTopicName@aSharedSubscriberName.

With this Topic name:

- The Queuing Service DataReader will only match the QueueProducer<aMessageType>
- The Queuing Service DataWriter will only match the QueueConsumer<aMessageType>

Figure 2.2 Queuing Service Entities and Topics

2.7 Sample Distribution to a Selected QueueConsumer

Queuing Service implements the logic that decides which QueueConsumer DataReader gets each sample. To distribute a sample to the selected QueueConsumer, the QueueConsumer DataReader uses a ContentFilteredTopic on the related_reader_guid. For example:
Interaction of Publish-Subscribe Entities with Queuing Service Entities

Queuing Service uses the `write_w_param()` operation on the SharedReaderQueue `DataReader` to set the `related_reader_guid` to the value specified in the filter expression of the selected `DataReader` (see Figure 2.3).

In Figure 2.3, when Queuing Service wants to send a sample to the first `DataReader`, it sets the field `related_reader_guid` in `WriteParams_t` to 0xFF. To send to the second `DataReader`, `related_reader_guid` is set to 0xEF.

2.8 Interaction of Publish-Subscribe Entities with Queuing Service Entities

A regular `DataReader` of Topic `aQueueTopicName@aSharedSubscriberName` will match a SharedReaderQueue `DataWriter` for the `SharedSubscriber aSharedSubscriberName` of Topic `aQueueTopicName`. However, Queuing Service will notice that the `DataReader`’s `SharedSubscriberName` is not set and interpret this to mean that it does not want to share the ReaderQueue. Instead, the `DataReader` wants traditional publish-subscribe access, which means it will get a copy of each sample that is sent to any of the `QueueConsumers`. See Figure 2.4.

This approach ensures that RTI DDS Spy, RTI Recording Service, and other such tools that observe data will continue to function without changes.

Note that the `QueueProducer` has a regular `DataWriter` of Topic `aQueueTopicName`. In addition to the SharedReaderQueue `DataReader`, it will also match any regular `DataReader` of that Topic.
Consequently, the regular Connext DDS tools (such as RTI DDS Spy and RTI Recording Service) will also receive the data sent by the QueueProducer.

Figure 2.4 Queuing Service Endpoint Matching with non-QueueConsumer DataReaders

2.9 Sample Lifecycle In Queuing Service

The samples received by a Queuing Service instance have a lifecycle described by the following states:

- **Enqueued**: The sample has been received by Queuing Service and has been stored in the SharedReaderQueue (persistent or in memory).
  
  In addition, if SharedReaderQueue replication is enabled, the sample must have been received and stored by a quorum of up-to-date replicas. See Chapter 7: High Availability.

- **Assigned**: The sample has been assigned to one of the QueueConsumers.
  
  In addition, if SharedReaderQueue replication is enabled, the selected QueueConsumer must have been communicated to the replicas and optionally confirmed by a quorum of up-to-date replicas. See Chapter 7: High Availability.

- **Sent**: The sample has been sent to the designated QueueConsumer.

- **Delivered**: The sample has been delivered to a QueueConsumer and the (application-level) acknowledgment of the successful delivery has been received from the QueueConsumer.
  
  In addition, if SharedReaderQueue replication is enabled, the delivery of the sample must have been communicated to the replicas. See Chapter 7: High Availability.
Sample Lifecycle In Queuing Service

- **Rejected**: The sample has been delivered to the selected QueueConsumer and the (application-level) acknowledgment from the QueueConsumer has been received with an indication that the message has been rejected.

- **Timed out**: The sample has been delivered to the selected QueueConsumer and it has not been (application-level) acknowledged from the QueueConsumer for a configurable maximum time (set using the tag `<response_timeout>` under `<queue_qos>/<redelivery>`), see Table 3.15, Queue QoS Tags.

- **Expired**: Indicates a sample that has exceeded a configurable maximum time to be held by Queuing Service. The sample lifespan can be configured per SharedReaderQueue or per QueueProducer:
  - To configure the sample lifespan per SharedReaderQueue, use the `<lifespan>` tag under `<queue_qos>`, see Table 3.15, Queue QoS Tags.
  - To configure the sample lifespan per QueueProducer, set the Lifespan QoS policy for the QueueProducer's `DataWriter`. This way all the samples sent by that QueueProducer will have a lifespan equal to the `writer_qos.lifespan.duration`. The lifespan per QueueProducer, when finite, takes precedence over the lifespan per SharedReaderQueue.

- **FailedDelivery**: Indicates a sample that has not been successfully delivered to any QueueConsumer after the maximum number of attempts (configured using the tag `<max_delivery_retries>` under `<queue_qos>/<redelivery>` for a SharedReaderQueue, see Table 3.15, Queue QoS Tags).

In addition to the state, each sample has a flag that indicates whether the sample may be a duplicate. This flag is set when Queuing Service sends a sample to a QueueConsumer but cannot ensure that no other QueueConsumer has processed it.

You can inspect the status of the duplicate flag in a received sample by inspecting the field flag in the SampleInfo. A sample may be a duplicate if the bit `DDS_REDECLIVERED_SAMPLE` is active.

**Table 2.1 State Transitions**

<table>
<thead>
<tr>
<th>State(s)</th>
<th>Transition Event to Next State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;Init&gt;</td>
<td>Sample is received by SharedReaderQueue <code>DataReader</code> and stored in the SharedReaderQueue. If SharedReaderQueue replication is enabled, sample is received and stored by a quorum of up-to-date replicas. Queuing Service sends an AppAck message to the QueueProducer.</td>
<td>Enqueued</td>
</tr>
<tr>
<td></td>
<td>Queuing Service decides which QueueConsumer should get the sample. If SharedReaderQueue replication is enabled, the selected QueueConsumer has been communicated to the replicas and optionally confirmed by a quorum of up-to-date replicas.</td>
<td>Assigned</td>
</tr>
</tbody>
</table>
2.10 Selecting a QueueConsumer for a Sample

*Queueing Service* implements the logic that decides which QueueConsumer gets each sample. This decision can be made according to different dispatch policies. To configure a dispatch policy and its properties, use the `<distribution>` tag under `<queue_qos>` (see Table 3.15, Queue QoS Tags).

### 2.10.1 Round-Robin Dispatch Policy without Explicit QueueConsumer Availability Feedback

This dispatch mode uses a round-robin approach to dispatch messages among the QueueConsumers that have acknowledged all previous messages sent to them up to a specified threshold. This dispatch mode does not require explicit feedback from the QueueConsumer.

For example, with a threshold of zero, samples are round-robin’ed among QueueConsumers that have acknowledged all previous samples that were sent to them. With a threshold of 2, samples are round-robin’ed among QueueConsumers that have acknowledged all samples sent to them except up to 2 samples (i.e., have acknowledged all, all but one, or all but two). With a threshold of UNLIMITED (-1), samples are round-robin’ed among all QueueConsumers, regardless of the number of outstanding unacknowledged samples in each one of them.
With this dispatch mode, the threshold is set per SharedReaderQueue using the property UNACKED_THRESHOLD. For example:

```xml
<distribution>
  <kind>ROUND_ROBIN</kind>
  <property>
    <element>
      <name>UNACKED_THRESHOLD</name>
      <value>-1</value>
    </element>
  </property>
</distribution>
```

### 2.10.2 Round-Robin Dispatch Policy with Explicit QueueConsumer Availability Feedback

This dispatch mode uses a QueueConsumer Availability Topic, which is published by the QueueConsumers and provides information about the capability of the QueueConsumer to process messages from Queuing Service. The round-robin will be done among the QueueConsumers that are available.

The ConsumerAvailability topic name is as follows: ConsumerAvailability@<SharedReaderQueueName>, where <SharedReaderQueueName> is <SharedReaderQueueTopicName>@<Shared-SubscriberName>.

The topic type is the following and can be found in <NDDSHOME>/resource/idl/QueuingServiceTypes.idl.

```c
struct ConsumerAvailability_t {
    GUID_t consumer_guid; //@key
    boolean reception_enabled;
    long unacked_threshold;
}; //@Extensibility EXTENSIBLE_EXTENSIBILITY
```

The type is registered with the following name: RTI::QueuingService::ConsumerAvailability_t.

A QueueConsumer can report its availability by updating the unacked_threshold and reception_enabled fields. The unacked_threshold field is equivalent to the threshold parameter described in Round-Robin Dispatch Policy without Explicit QueueConsumer Availability Feedback (Section 2.10.1) but it can be set per QueueConsumer.

In addition, a QueueConsumer can indicate that it does not want to receive any samples from Queuing Service by setting the field reception_enabled to DDS_BOOLEAN_FALSE.

The field consumer_id must be used to identify the QueueConsumer that sends the Availability sample. This field must be the same value that was used to set the QueueConsumer ContentFilteredTopic described in Sample Distribution to a Selected QueueConsumer (Section 2.7).

By default, when using ROUND_ROBIN policy, a SharedReaderQueue does not create a DataReader to receive availability updates from a QueueConsumer. To enable that behavior, set the property ALLOW_CONSUMER_FEEDBACK to 1. For example:

```xml
<distribution>
  <kind>ROUND_ROBIN</kind>
  <property>
    <element>
      <name>UNACKED_THRESHOLD</name>
      <value>-1</value>
    </element>
  </property>
</distribution>
```
Notice that in a SharedReaderQueue with the previous configuration it is possible to have some QueueConsumers reporting availability through the new topic and some QueueConsumers not reporting availability and using the configuration threshold under `<distribution>`.

### 2.11 Sending a Reply from QueueConsumer to QueueProducer

*Queueing Service* also supports a request-reply communication model in which a requester application sends a sample to a SharedReaderQueue, and a replier application receives the sample from the SharedReaderQueue and returns a response to the requester application.

Realizing the request-reply communication model requires creating a new SharedReaderQueue that will be used to send responses from the replier application to the requester application (see Figure 2.5).

**Figure 2.5 Request-Reply Communication Model**

![Diagram of Request-Reply Communication Model](image)

#### 2.11.1 Requester Identification

In a request-reply pattern, requests must uniquely identify the associated QueueProducer so that each reply sample can be unambiguously delivered to the requester application that sent the associated request. To identify a QueueProducer, you can use the source GUID.
The source_guid consists of a GUID; it can be set per sample using the source_guid field in the WriteParams_t parameter provided to the QueueProducer's DataWriter write_w_params() operation.

If you do not want to set the source GUID of a sample, the QueueProducer's DataWriter will assign it automatically to be equal to the DataWriter's virtual GUID.

In general, you should always assign the source GUID when sending requests. Otherwise, the requester application will not be robust to potential restarts. If the source GUID is different every time the requester application restarts, there may be responses that get lost since Queuing Service will not know how to route them to the proper requester application.

### 2.11.2 Request-Reply Correlation

When the replier application receives a request sample from Queuing Service, it must extract the source GUID and the sample identity in order to send them back as part of the reply to the requester application. This allows requests and replies to be correlated.

The replier application can extract the identity of a request from the fields related_original_publication_virtual_guid and related_original_publication_virtual_sequence_number in the SampleInfo associated with the request sample (see Figure 2.6).

Figure 2.6 Request Generation

The replier application can extract the source GUID of a request from the field related_source_guid of the SampleInfo associated with the request sample (see Figure 2.6).

Once the replier application extracts the request sample identity and source GUID from the request SampleInfo, it must attach them to the reply sample as follows:

- The sample identity will be set using the field related_sample_identity in the WriteParams_t parameter provided to the DataWriter's write_w_params() operation.
- The source GUID will be set using the field related_source_guid in the WriteParams_t parameter provided to the DataWriter's write_w_params() operation.
When the requester application receives the reply, it can associate the reply with the corresponding request by inspecting the related_original_publication_virtual_guid and related_original_publication_virtual_sequence_number fields in the SampleInfo associated with the reply sample (see Figure 2.6).

### 2.11.3 Sending the Reply Sample to the Associated Requester

To guarantee that a reply sample is only distributed to the right Requester, the DataReader in the Requester must use a ContentFilteredTopic on the related_source_guid, where the value is set to the source GUID associated with the request. For the example in Figure 2.6, the filter would be:

```
(related_source_guid.value = &hex(<SGUIDm>))
```

Alternatively, you can set the filter in the related_reader_guid, as follows:

```
(related_reader_guid.value = &hex(<SGUIDm>))
```

### 2.11.4 QueueRequester Wrapper

To simplify the use and configuration of the DataReader and DataWriter in the requester application, Connext DDS provides an abstraction, QueueRequester<MessageRequestType, MessageReplyType>, which wraps the DataReader and DataWriter usage and provide additional services such as an operation to wait for the response for a given request.

For more information, see Chapter 8: Queuing Service Wrapper API.

In this release, the QueueRequester wrapper API is only supported in the .NET API.

### 2.11.5 QueueReplier Wrapper

To simplify the use and configuration of the DataReader and DataWriter in the replier application, Connext DDS provides an abstraction, QueueReplier<MessageRequestType, MessageReplyType>, which wraps the DataReader and DataWriter usage.

For more information, see Chapter 8: Queuing Service Wrapper API.

In this release, the QueueReplier wrapper API is only supported in the .NET API.

### 2.12 Dead-Letter Queues

Queuing Service provides support for dead-letter queues. A dead-letter queue is a SharedReader-Queue to which other SharedReaderQueues can send messages that for some reason could not be successfully delivered and processed.

Queuing Service supports the definition of one dead-letter queue per SharedSubscriber by using the XML tag `<dead_letter_shared_reader_queue>`. The dead-letter queue has two limitations compared with a regular queue:

1. It cannot have a `<reply_type>`.
2. It cannot have a `<type_name>`.

The type associated with the samples in a dead-letter queue is `DeadLetter_t`, defined as follows:

```c
enum UndeliveredReasonKind {
    LIFESPAN_UNDELIVERED_REASON_KIND,
    MAX_RETRIES_UNDELIVERED_REASON_KIND
}
struct GUID_t {
```
Dead-Letter Queues

```c
    octet value[16];

    struct SequenceNumber_t {
        long high;
        unsigned long low;
    };

    struct SampleIdentity_t {
        GUID_t writer_guid;
        SequenceNumber_t sequence_number;
    };

    struct SampleBuffer_t {
        sequence<octet> value;
    };

    struct DeadLetter_t {
        string queue_name;
        SampleIdentity_t sample_identity;
        UndeliveredReasonKind undelivered_reason;
        SampleBuffer_t sample_buffer;
    }; //@Extensibility EXTENSIBLE_EXTENSIBILITY
```

You can find the IDL file that defines the DeadLetter types in `<NDDSHOME>/resource/idl/QueuingServiceTypes.idl`.

The `queue_name` has the format `<aQueueTopicName>@<aSharedSubscriberName>`.

The `sample_identity` contains the identity of the undelivered sample.

The `sample_buffer` contains the sample data in serialized form with CDR representation. To deserialize the sample data, use the following operations:

- C: `FooTypeSupport_deserialize_data_from_cdr_buffer()`
- C++: `FooTypeSupport::deserialize_data_from_cdr_buffer()`
- Java: `FooTypeSupport.get_instance().deserialize_from_cdr_buffer()`
- C++/CLI: `FooTypeSupport::deserialize_data_from_cdr_buffer()`
- C#: `FooTypeSupport.deserialize_data_from_cdr_buffer()`

For additional information on these deserialization operations, see the Connext DDS API Reference HTML documentation.

The `undelivered_reason` is an enumeration describing why the sample was not delivered. There are two possible reasons:

- The lifespan expired for the sample.
- The sample exceeded the maximum number of redelivery retries.

For more information on why a sample may be undelivered, see Sample Lifecycle In Queuing Service (Section 2.9).

By default, SharedReaderQueues do not send undelivered samples to the dead-letter queue. To enable this behavior, you must use the attribute `dead_letter_queue` in the `<shared_reader_queue>` tag. This attribute must be set to the name of the dead-letter queue in the configuration file.
2.13 Detecting the Presence of a SharedReaderQueue

You can detect the existence of a SharedReaderQueue for a given QueueProducer or QueueConsumer by monitoring the matched subscriptions associated with the QueueProducer’s DataWriter and the matched publications associated with the QueueConsumer’sDataReader.

The PublicationBuiltinTopicData and SubscriptionBuiltinTopicData include a field called service, which, in the case of a Queuing Service DataWriter or DataReader, is set to QUEUING_SERVICE_QOS.

Since the durability of the QueueProducer DataWriter is normally set to VOLATILE, to guarantee that the initial samples are received by a Queuing Service instance, the application should check that there is a match between a QueueProducer DataWriter and a SharedReaderQueueDataReader before starting to publish samples.

For convenience and ease of use, the wrapper APIs offer methods to detect when there are matching SharedReaderQueue for QueueProducers, QueueConsumers, QueueRequesters, and QueueRepliers. See Chapter 8: Queuing Service Wrapper API.

2.14 Queuing Service Persistency

By default, both the service state and the SharedReaderQueues samples are kept in memory.

For fault tolerance, and to preserve the current configuration, Queuing Service can be configured to persist its configuration, as well as the SharedReaderQueues samples to disk.

Figure 2.7 Service State Persistency

2.14.1 Service State Persistency

The configuration of a Queuing Service instance is dynamic. Once the service is bootstrapped from a configuration file in XML format or remotely by getting the configuration from other
Queuing Service instances, the configuration can be changed at run time by sending remote commands to the service (see Chapter 5: Administering Queuing Service from a Remote Location). For example, you may decide to add a new SharedReaderQueue or to remove a SharedReaderQueue.

You can choose to persist the configuration to disk each time it changes by setting the `<kind>` tag within `<queuing_service>/<service_qos>/<persistence>` to PERSISTENT (see Table 3.15, Queue QoS Tags).

The location of the file where the configuration is persisted, as well as the properties of the storage process, can be configured using the `<filesystem>` tag under `<queuing_service>/<persistence_settings>` (see Configuring Persistence Settings (Section 3.3.5)).

When Queuing Service is restarted, it will look for its persisted configuration using the following values:

- Command-line option `-appName` (see Queuing Service runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin. There are four ways to start Queuing Service: (Section ))
- XML tag values `<directory>` and `<file_prefix>` under `<persistence_settings>/<filesystem>`

If the persisted configuration is found and the service is configured from a XML file, the persisted configuration will be used to configure the service instance. In that case, the input XML file is only used to find the location of the persistent storage and configure the storage process. If the persisted configuration is not found, the service will be initialized using the input XML file.

When the service configuration is obtained remotely using the command-line option `-cfgRemote` (see Table 4.1, RTI Queuing Service Command-Line Options), any persisted configuration will be dropped and the service will always be initialized using the remote XML configuration.

The location and name of the file where the configuration is persisted is as follows:

```
[directory]/[prefix]service@[appName].db
```

Where:

- `[directory]` is configured using the tag `<directory>` under `<persistence_settings>/<filesystem>`
- `[prefix]` is configured using the tag `<file_prefix>` under `<persistence_settings>/<filesystem>`
- `[appName]` is configured using the command-line parameter `-appName`.

### 2.14.2 SharedReaderQueue Persistency

A SharedReaderQueue can be configured to persist the undelivered samples into disk by setting the XML tag `<kind>` within `<shared_reader_queue>/<queue_qos>/<persistence>` to PERSISTENT (see Table 3.15, Queue QoS Tags).

Queuing Service provides two different PERSISTENT implementations:

- **Without In-Memory State**: In this mode, the metadata and user data associated with the SharedReaderQueue’s samples is kept only on disk. Every time the metadata or user data is used, Queuing Service reads it from disk.

- **With In-Memory State**: In this mode, the metadata for the SharedReaderQueue’s samples is always kept both on disk and in memory. The sample’s user data is kept in memory and on disk only when:
• Its serialized size is smaller than the threshold set using the tag `<domain_participant>/<memory_management>/<sample_buffer_min_size>` (see Memory Management for a Sample (Section 2.15.2)).

• `<domain_participant>/<memory_management>/<sample_buffer_trim_to_size>` is set to false (see Memory Management for a Sample (Section 2.15.2)).

PERSISTENT SharedReaderQueues with in-memory state introduce significant performance improvements because the sample metadata, and in some cases the sample user data, does not need to be accessed from disk. The disadvantage is that the number of samples on the SharedReaderQueue is limited by the available memory, as the service needs to keep some state per sample in memory.

To configure a PERSISTENT SharedReaderQueue to keep the sample state in-memory (the default configuration), you must set the XML tag `<in_memory_state>` under `<queue_qos>/<persistence>` to true.

Samples are persisted before Queuing Service sends an application-level acknowledgement (AppAck) message to the QueueProducer DataWriter indicating successful processing of the sample.

Like with the service configuration, the location of the file(s) where the SharedReaderQueue's samples are persisted, as well as the properties of the storage process, can be configured using the `<filesystem>` tag under `<queuing_service>/<persistence_settings>` (see Configuring Persistence Settings (Section 3.3.5)).

When a SharedReaderQueue is created, the service will locate its persisted samples using the following values:

- Command-line option `-appName` (see Queuing Service runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin`. There are four ways to start Queuing Service: (Section ))
- XML tag values `<directory>` and `<file_prefix>` under `<persistence_settings>/<filesystem>`
- The SharedSubscriber's name configured using the name attribute in `<shared_subscriber>`
- The SharedReaderQueue's topic name configured using the XML tag value `<topic_name>` under `<shared_reader_queue>`
- The DomainParticipant's domain ID configured using the XML tag value `<domain_id>` under `<domain_participant>`

If the samples are found, the SharedReaderQueue will be initialized with them.

The location and name of the file where the SharedReaderQueue's samples are persisted is as follows:

- Without in-memory state:
  `[directory]/[prefix] [topicName][sharedSubscriberName][domainId][appName].db`

- With in-memory state:
  For data:
  `[directory]/[prefix] [topicName][sharedSubscriberName][domainId][appName]_d[fileIndex].db`
  For metadata:
  `[directory]/[prefix] [topicName][sharedSubscriberName][domainId][appName]_m[fileIndex].db`

Where:
2.15 SharedReaderQueue Resource Management

Queuing Service provides fine-grained control over the resources (memory and disk) associated with the samples in a SharedReaderQueue. It provides ways to monitor when the space taken by the samples in a SharedReaderQueue goes above or below configurable watermarks and when the SharedReaderQueue fills up. Finally, it also provides a way to configure the SharedReaderQueue behavior when a new sample arrives and the SharedReaderQueue is full.

2.15.1 Maximum SharedReaderQueue Size

The maximum size of a SharedReaderQueue can be configured based on number of samples, number of bytes in memory, or both.

2.15.1.1 Initial and Maximum Number of Samples

The tag `<resource_limits>` under `<shared_reader_queue>/<queue_qos>` can be used to configure the initial and maximum number of samples in a SharedReaderQueue (see Table 3.15, Queue QoS Tags) as well as if dynamic allocations are allowed and how they occur.

Example:
<resource_limits>
  <queue_allocation_settings>
    <initial_count>10</initial_count>
    <max_count>LENGTH_UNLIMITED</max_count>
    <incremental_count>-1</incremental_count>
  </queue_allocation_settings>
</resource_limits>

In the above example:

- **initial_count**: Queuing Service will pre-allocate ten queue samples in advance.
- **max_count**: The maximum number of samples that the queue can hold is UNLIMITED.
- **incremental_count**: As additional samples are needed, Queuing Service will double the amount of extra memory allocated each time memory is needed.

Ranges:

- **initial_count**: positive number and < max_count
- **max_count**: LENGTH_UNLIMITED or positive number
- **incremental_count**: -1 (double) or positive number

Defaults:

- **initial_count**: 1
- **max_count**: LENGTH_UNLIMITED
- **incremental_count**: -1

When **max_count** is exceeded, the behavior of a SharedReaderQueue when new samples are received can be configured using **<replacement_policy>** under **<resource_limits>**. See Sample Replacement Policy (Section 2.15.4).

### 2.15.1.2 Maximum Number of Bytes in Memory

The tag **<resource_limits>** under **<shared_reader_queue>/<queue_qos>** can also be used to configure the maximum size of a SharedReaderQueue based on the number of bytes required to store the samples in-memory. For example:

```
<resource_limits>
  <queue_allocation_settings>
    <max_in_memory_bytes>1000000</max_in_memory_bytes>
  </queue_allocation_settings>
</resource_limits>
```

In the above example the size required to store the SharedReaderQueue samples in-memory cannot exceed 1,000,000 bytes. Notice that if the SharedReaderQueue does not have any samples and the size of a new sample exceeds 1,000,000 bytes, this sample will be stored in the SharedReaderQueue. Therefore, it is possible to go beyond 1,000,000 bytes when the SharedReaderQueue is empty.

The configuration parameter **max_in_memory_bytes** includes both the sample metadata and the sample user data. The parameter does not take into account the SharedReaderQueue metadata and the preallocated samples (metadata and user data) that are not currently used.

If both **<max_count>** and **<max_in_memory_bytes>** are set to a finite number, the maximum size of the SharedReaderQueue will be limited by the limit that is reached first.

**<max_in_memory_bytes>** is ignored for PERSISTENT SharedReaderQueues where the state is not kept in-memory.

Ranges:
- **max_in_memory_bytes**: LENGTH_UNLIMITED or positive number

  **Defaults:**
  - **max_in_memory_bytes**: LENGTH_UNLIMITED

#### 2.15.2 Memory Management for a Sample

For every sample in a SharedReaderQueue, *Queueing Service* will use a buffer to store the content of the sample in serialized form. The memory for that buffer may come from a pre-allocated pool of buffers or may be dynamically allocated from the heap upon sample reception. This behavior is controlled per `<domain_participant>` using the XML tag `<memory_management>` (see Table 3.10, DomainParticipant Tags), which affects all the SharedReaderQueues within the `<domain_participant>`.

For example:

```xml
<memory_management>
  <sample_buffer_min_size>16000</sample_buffer_min_size>
  <sample_buffer_trim_to_size>true</sample_buffer_trim_to_size>
</memory_management>
```

In the above example:

- **sample_buffer_min_size**: If the serialized size of an incoming sample is smaller or equal to 16000 bytes, *Queueing Service* will use a pre-allocated buffer from a pool to hold the sample. The initial and maximum number of buffers in the pool as well as the pool’s growth policy is configured using the XML tag `<resource_limits>` under `<shared_reader_queue>/<queue_qos>`. When the serialized size of the incoming sample is greater than 16,000 bytes, *Queueing Service* will allocate the buffer from the heap dynamically upon sample reception.

- **sample_buffer_trim_to_size**: For dynamically allocated buffers *Queueing Service* will release the memory after the sample is removed from the SharedReaderQueue.

For more information on `<memory_management>` and its default values, see Table 3.10, DomainParticipant Tags.

#### 2.15.3 High and Low Watermarks

The tag `<queue_watermark_settings>` under `<shared_reader_queue>/<queue_qos>/resource_limits` can be used to configure high and low watermarks in a SharedReaderQueue (see Table 3.15, Queue QoS Tags). Watermarks are expressed as a percentage with respect to the maximum number of samples or maximum number of bytes allowed in the SharedReaderQueue. For example:

```xml
<resource_limits>
  <queue_allocation_settings>
    <max_count>1000</max_count>
    <max_in_memory_bytes>10000000</max_in_memory_bytes>
  </queue_allocation_settings>
  <queue_watermark_settings>
    <high_watermark>90</high_watermark>
    <low_watermark>10</low_watermark>
  </queue_watermark_settings>
</resource_limits>
```

In the above example, the high watermark of 90% corresponds to 900 samples (9,000,000 bytes) and the low watermark of 10% corresponds to 100 samples (1,000,000 bytes).
An application can monitor if the number of samples in a SharedReaderQueue go over the high watermark or below the low watermark by retrieving the SharedReaderQueue status using the remote administration command Get SharedReaderQueue Status (see Get SharedReaderQueue Status (Section 5.4.4)) or by subscribing to the SharedReaderQueue status monitoring topic (see Chapter 6: Publish-Subscribe Monitoring of Queuing Service from a Remote Location).

The SharedReaderQueueStatus type used to provide the status of a SharedReaderQueue can be found <NDDSHOME>/resource/idl/QueuingServiceTypes.idl:

```
struct SharedReaderQueueStatus {
    ...  
    unsigned long long high_watermark_count;
    unsigned long long low_watermark_count;
    unsigned long long queue_full_count;
    ...  
    unsigned long long high_watermark_count_change;
    unsigned long long low_watermark_count_change;
    unsigned long long queue_full_count_change;
    ...  
}; //@Extensibility MUTABLE_EXTENSIBILITY
```

Where:

- **high_watermark_count**: Number of times that the SharedReaderQueue went over the high watermark since the service started.
- **low_watermark_count**: Number of times that the SharedReaderQueue went below the low watermark since the service started.
- **high_watermark_count_change**: Number of times that the SharedReaderQueue has gone over the high watermark since the last remote administration command to retrieve the status of the SharedReaderQueue.
- **low_watermark_count_change**: Number of times that the SharedReaderQueue has gone below the low watermark since the last remote administration command to retrieve the status of the SharedReaderQueue.

Notice that it is also possible to monitor how many times the SharedReaderQueue filled up by inspecting the fields `queue_full_count` and `queue_full_count_change`.

### 2.15.4 Sample Replacement Policy

The tag `<replacement_policy>` under `<shared_reader_queue>/<queue_qos>/resource_limits` can be used to configure a SharedReaderQueue behavior when it is full and a new sample is received. For example:

```
<resource_limits>
  <replacement_policy>
    <kind>REJECT_WITHOUT_REPLACEMENT</kind>
  </replacement_policy>
</resource_limits>
```

In the above example, a new incoming sample will be rejected if there is no space for it in the SharedReaderQueue. When a sample is rejected, and if `<app_ack_sample_to_producer>` is set to true for the SharedReaderQueue, *Queuing Service* will send an AppAck message to the QueueProducer with a payload byte set to 0.

This version of *Queuing Service* supports two kinds of replacement policies:

- **REJECT_WITHOUT_REPLACEMENT**: New samples are rejected when the SharedReaderQueue is full.
High Availability

- **WAIT WITHOUT REPLACEMENT**: New samples are kept in the SharedReaderQueue'sDataReader cache until they can be added to the SharedReaderQueue.

  **Default:**

  - **kind**: REJECT WITHOUT REPLACEMENT

  Notice that the WAIT WITHOUT REPLACEMENT replacement kind allows you to implement a flow-control mechanism with the QueueProducer's DataWriter in which the DataWriter's `write()` operation will block if new samples cannot be added to the SharedReaderQueue.

  To achieve this behavior:

  - The SharedReaderQueue's DataReader's cache must have a finite size. This can be done by configuring `<shared_reader_queue>/<datareader_qos>/<resource_limits>/<max_samples>` to a finite number.

  - The QueueProducer's DataWriter's send window size must be a finite value. This can be done by configuring `<datawriter_qos>/<protocol>/rtps_reliable_writer/<max_send_window_size>`.

  If a new sample arrives to the SharedReaderQueue and there is no space for it in the SharedReaderQueue's DataReader cache, the sample will be rejected by Connext DDS. The QueueProducer's DataWriter will not be able to mark that sample or any subsequent samples as acknowledged and eventually it will block after its send window fills up.

---

### 2.16 High Availability

For high availability, you can configure Queuing Service to replicate both the content of the SharedReaderQueues and the service configuration.

By default, SharedReaderQueues within a Queuing Service instance are not replicated. SharedReaderQueues can optionally be replicated across multiple instances of Queuing Service running on the same or different nodes.

By default, the service configuration is not replicated. The service configuration can optionally be replicated across multiple instances of Queuing Service running in the same or different nodes.

For more information on SharedReaderQueues and service configuration replication, see Chapter 7: High Availability.

---

### 2.17 Remote Administration

You can control Queuing Service remotely by sending commands through a special topic. Any Connext application can be implemented to send these commands and receive their corresponding responses.

These remote administration commands will allow you to:

- Create SharedReaderQueues
- Delete SharedReaderQueues
- Flush SharedReaderQueues
- Get SharedReaderQueues status
- Get service data
Get samples from a SharedReaderQueue

For more information on remote administration, see Chapter 5: Administering Queuing Service from a Remote Location.

### 2.18 Queuing Service Monitoring

With *Queuing Service*, you can monitor the status of the service and its SharedReaderQueues using request-reply or publish-subscribe communication patterns.

Request-reply monitoring is done by issuing remote administration commands that retrieve the status of the different entities in the service. See Chapter 5: Administering Queuing Service from a Remote Location.

Publish-subscribe monitoring is done by subscribing to monitoring topics. See Chapter 6: Publish-Subscribe Monitoring of Queuing Service from a Remote Location.
Chapter 3 Configuring Queuing Service

This chapter describes how to configure Queuing Service. For installation instructions, please see the Queuing Service Getting Started Guide.

Queuing Service is configured using a configuration in XML format. There are three different ways to provide the initial configuration to Queuing Service:

- **Configuration file:** The file(s) can be implicit or explicit using the -cfgFile command-line option (see How to Load the XML Configuration from a File (Section 3.1)).

- **Database:** The Queuing Service configuration can be persisted and restored from disk by enabling service state Persistency (see Service State Persistency (Section 2.14.1)).

- **Remote configuration:** Queuing Service can be set up to obtain its initial configuration remotely from a different Queuing Service instance by using the -cfgRemote command-line option (see Table 4.1).

Before reading this chapter, you should be familiar with Terms to Know (Section 2.1).

This chapter describes:

- How to Load the XML Configuration from a File (Section 3.1)
- XML Syntax and Validation (Section 3.2)
- XML Tags for Configuring Queuing Service (Section 3.3)

### 3.1 How to Load the XML Configuration from a File

Queuing Service loads its XML configuration file(s) from multiple locations. This section presents the various approaches, listed in load order.

The first three locations only contain QoS Profiles and are inherited from Connext DDS (see Chapter 15 in the RTI Connext DDS Core Libraries User’s Manual).

- `<NDDSHOME>/resource/xml/NDDS_QOS_PROFILES.xml`
  
  This file contains the Connext DDS default QoS values; it is loaded automatically if it exists. (First to be loaded)

- File in NDDS_QOS_PROFILES
  
  The files (or XML strings) separated by semi-colons referenced in this environment variable are loaded automatically.

- `<working directory>/USER_QOS_PROFILES.xml`
  
  This file is loaded automatically if it exists.
The next locations are specific to Queuing Service:

- `<NDDSHOME>/resource/xml/RTI_QUEUING_SERVICE.xml`
  This file contains the default Queuing Service configuration; it is loaded if it exists. RTI_QUEUING_SERVICE.xml defines a service with an empty SharedSubscriber and with administration enabled.

- `<working_directory>/USER_QUEUING_SERVICE.xml`
  This file is loaded automatically if it exists.

- File specified using the command line parameter `-cfgFile`
  The command-line option `-cfgFile` (see Table 4.1, RTI Queuing Service Command-Line Options) can be used to specify a configuration file.

An example configuration file is seen below. You will learn the meaning of each line as you read the rest of this chapter.

**Example XML Configuration File**

```xml
<?xml version="1.0"?><dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="rti_queuing_service.xsd">
  <types>
    <struct name="Foo">
      <member type="string" stringMaxLength="255" name="message"/>
    </struct>
    <struct name="Bar">
      <member type="string" stringMaxLength="255" name="message"/>
    </struct>
  </types>
  <queuing_service name="QueuingService_1">
    <administration>
      <domain_id>56</domain_id>
    </administration>
    <domain_participant name="DomainParticipant_1">
      <domain_id>57</domain_id>
      <shared_subscriber name="SharedSubscriber_1">
        <session_settings>
          <session name="Session_1"/>
        </session_settings>
        <dead_letter_shared_reader_queue name="DeadLetter_1">
          <topic_name>DeadLetter</topic_name>
        </dead_letter_shared_reader_queue>
        <shared_reader_queue session="Session_1" dead_letter_queue="DeadLetter_1">
          <topic_name>HelloWorld</topic_name>
          <type_name>Foo</type_name>
          <reply_type>Bar</reply_type>
          <queue_qos>
            <distribution>
```
<kind>ROUND_ROBIN</kind>
</distribution>

{lifespan>
  <duration>
    <sec>120</sec>
    <nanosec>0</nanosec>
  </duration>
</lifespan>

<redelivery>
  <reponse_timeout>
    <duration>
      <sec>10</sec>
      <nanosec>0</nanosec>
    </duration>
  </reponse_timeout>
  <max_delivery_retries>10</max_delivery_retries>
</redelivery>

</queue_qos>
</shared_reader_queue>
</shared_subscriber>
</domain_participant>
</queuing_service>
</dds>

## 3.2 XML Syntax and Validation

The XML configuration file must follow these syntax rules:

- The syntax is XML; the character encoding is UTF-8.
- Opening tags are enclosed in `<>`; closing tags are enclosed in `</>`.
- A tag value is a UTF-8 encoded string. Legal values are alphanumeric characters. Queuing Service’s parser will remove all leading and trailing spaces from the string before it is processed. For example, "<tag> value </tag>" is the same as "<tag>value</tag>".
- All values are case-sensitive unless otherwise stated.
- Comments are enclosed as follows: `<!-- comment -->`.
- The root tag of the configuration file must be `<dds>` and end with `</dds>`.

Queuing Service provides an XSD file that describes the format of the XML content. We recommend including a reference to this file in the XML file that contains the Queuing Service configuration—this provides helpful features in code editors such as Visual Studio and Eclipse, including validation and auto-completion while you are editing the XML file.

The XSD definition of the XML elements is in `<NDDSHOME>/resource/schema/rti_queuing_service.xsd`.

To include a reference to the XSD document in your XML file, use the attribute `xsi:noNamespaceSchemaLocation` in the `<dds>` tag. For example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="<Queuing Service installation directory>/resource/schema/rti_queuing_service.xsd">
```
3.3 XML Tags for Configuring Queuing Service

This section describes the XML tags you can use in a Queuing Service configuration file. The following diagram and Table 3.1 describe the top-level tags allowed within the root <dds> tag.

![Diagram of XML tags for configuring Queuing Service]

Table 3.1  Top-Level Tags in the Configuration File

<table>
<thead>
<tr>
<th>Tags within &lt;dds&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;queuing_service&gt;</td>
<td>Specifies a Queuing Service configuration. See Configuring Queuing Service (Section 3.3.2).</td>
<td>1 or more (required)</td>
</tr>
<tr>
<td>&lt;qos_library&gt;</td>
<td>Specifies a QoS library and profiles. The contents of this tag are specified in the same manner as for a Connext DDS QoS profile file. See Chapter 15 in the RTI Connext DDS Core Libraries User’s Manual.</td>
<td>0 or more</td>
</tr>
<tr>
<td>&lt;types&gt;</td>
<td>Defines types that can be used by Queuing Service. See Configuring Queuing Service Types (Section 3.3.1).</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

3.3.1 Configuring Queuing Service Types

Queuing Service allows users to provide type definitions for a SharedReaderQueue using two different mechanisms:

- Type definition in the XML configuration file
- Type discovery
To define and use a type in your XML configuration file:

1. Define your type within the `<types>` tag. (This is one of the top-level tags, see Table 3.1, Top-Level Tags in the Configuration File.)

2. Refer to it using its fully qualified name in the SharedReaderQueues that will use it.

For example:

```xml
<?xml version="1.0"?>
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="rti_queuing_service.xsd">
    <types>
        <struct name="Foo">
            <member type="string" stringMaxLength="255" name="message"/>
        </struct>

        <struct name="Bar">
            <member type="string" stringMaxLength="255" name="message"/>
        </struct>
    </types>

    <queuing_service name="QueuingService_1">
        ...
        <domain_participant name="DomainParticipant_1">
            <domain_id>57</domain_id>

            <shared_subscriber name="SharedSubscriber_1">
                <session_settings>
                    <session name="Session_1" />
                </session_settings>

                <dead_letter_shared_reader_queue name="DeadLetter_1"
                    session="Session_1">
                    <topic_name>DeadLetter</topic_name>
                </dead_letter_shared_reader_queue>

                <shared_reader_queue session="Session_1"
                    dead_letter_queue="DeadLetter_1">
                    <topic_name>HelloWorld</topic_name>
                    <type_name>Foo</type_name>
                    <reply_type>Bar</reply_type>
                </shared_reader_queue>
            </shared_subscriber>
        </domain_participant>
    </queuing_service>
</dds>
```

When types are defined in XML, Queuing Service is registering them with the underlying DDS DomainParticipant using as the registration name the fully qualified name of the type under the `<type>` tag.

If you refer to types that are not defined in the configuration file, Queuing Service has to discover the type representation (e.g., a typeobject). A SharedReaderQueue cannot be instantiated without the type representation information.
3.3.2 Configuring Queuing Service

A configuration file must have at least one `<queuing_service>` tag, which is used to configure an execution of Queuing Service. A configuration file may contain multiple `<queuing_service>` tags.

When you start Queuing Service, you can specify which `<queuing_service>` tag to use to configure the service using the `-cfgName` command-line parameter.

For example:

```xml
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="rti_queuing_service.xsd">

  ...
  <queuing_service name="QueuingService_1">
    ...
  </queuing_service>
  ...
  <queuing_service name="QueuingService_2">
    ...
  </queuing_service>

</dds>
```

Starting Queuing Service with the following command will use the `<queuing_service>` tag with the name `QueuingService_1`:

```bash
Queuingservice -cfgFile example.xml -cfgName QueuingService_1
```

Because a configuration file may contain multiple `<queuing_service>` tags, one file can be used to configure multiple Queuing Service executions.

Table 3.2, Queuing Service Tags describes the tags allowed within a `<queuing_service>` tag. Notice that the `<domain_participant>` tag is required.
### XML Tags for Configuring Queuing Service

#### Table 3.2 Queuing Service Tags

<table>
<thead>
<tr>
<th>Tags within &lt;queuing_service&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;administration&gt;</td>
<td>Enables and configures remote administration. See Configuring Administration (Section 3.3.3) and Chapter 5: Administering Queuing Service from a Remote Location.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;domain_participant&gt;</td>
<td>For each &lt;domain_participant&gt; tag, Queuing Service creates one Domain-Participant to communicate over DDS. SharedSubscribers are defined within a &lt;domain_participant&gt;. See Configuring DomainParticipants (Section 3.3.6).</td>
<td>1 or more (required)</td>
</tr>
<tr>
<td>&lt;monitoring&gt;</td>
<td>Enables and configures general remote Pub/Sub monitoring. See Configuring Monitoring (Section 3.3.4).</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;persistence_settings&gt;</td>
<td>Configures the storage settings that are used to persist the service state as well as the SharedReaderQueues samples. See Configuring Persistence Settings (Section 3.3.5).</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| <replication_settings>       | Configures the default settings for the replication protocol for SharedReaderQueues and configuration. These settings can be overridden by the settings under:  
  - <shared_reader_queue_replication> under <service_qos>  
  - <configuration_replication> under <service_qos>  
  - <replication> under <queue_qos>  
  **Important:** Using this tag does not enable replication. To enable replication, set:  
  - <shared_reader_queue_replication> under <service_qos> or <replication> under <queue_qos> for SharedReaderQueues  
  - <configuration_replication> under <service_qos> for configuration  
  See Chapter 7: High Availability. | 0 or 1 |
| <service_qos>                | Configures the QoS for the service. See Table 3.3, Service QoS Tags | 0 or 1 |
| <statistics>                 | Configures the statistics-gathering process for publish-subscribe and request-reply monitoring. See Configuring Monitoring (Section 3.3.4). | 0 or 1 |

#### 3.3.3 Configuring Administration

You can create a Connext DDS application that can remotely control Queuing Service. The <administration> tag is used to enable remote administration and configure its behavior.

By default, remote administration is turned off in Queuing Service for security reasons. A remote administration section is not required in the configuration file.

For example:

```xml
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="rti_queuing_service.xsd">
  <queuing_service name="QueuingService_1">
    <administration>
      <domain_id>55</domain_id>
    </administration>
  </queuing_service>
</dds>
```
When remote administration is enabled, Queuing Service will create a DomainParticipant, Publisher, Subscriber, DataWriter, and DataReader. These entities are used to receive commands and send responses. You can configure these entities with QoS tags within the <administration> tag.

Table 3.4, Remote Administration Tags lists the tags allowed within <administration> tag. Notice that the <domain_id> tag is required.

For more details, please see Chapter 5: Administering Queuing Service from a Remote Location.

**Note:** The command-line options used to configure remote administration take precedence over the XML configuration.

### 3.3.3.1 Configuring Memory Management for a CommandReply Buffer

The <memory_management> tag under <administration> controls how Queuing Service allocates memory for the string_body or octet_body buffer in a CommandReply.

For example:

```xml
<memory_management>
  <sample_buffer_min_size>16000</sample_buffer_min_size>
  <sample_buffer_trim_to_size>true</sample_buffer_trim_to_size>
</memory_management>
```
sample_buffer_min_size: If the size required for the buffer of a CommandReply is smaller or equal to this value, Queuing Service will use a pre-allocated buffer. The size of this buffer is equal to this value.

If the size required for the buffer of a CommandReply is greater than this value, Queuing Service will allocate the buffer from the heap dynamically upon reply generation.

sample_buffer_trim_to_size: This value controls what to do with the CommandReply buffer that is dynamically allocated. When true, the buffer will be released when the corresponding reply is sent. When false, the buffer is retained for future responses. It may be released later on, but only to be replaced by a larger buffer.

Ranges:

- sample_buffer_min_size: -1 (2 GB, the maximum size of a CommandReply) or a positive number.
- sample_buffer_trim_to_size: true or false

Defaults:

- sample_buffer_min_size: 32768
- sample_buffer_trim_to_size: false
3.3.4 Configuring Monitoring

With Queuing Service, you can monitor the status of the service and its SharedReaderQueues using request-reply or publish-subscribe communication patterns.

Request-reply monitoring is done by issuing remote administration commands that retrieve the status of the different entities in the service. See Chapter 5: Administering Queuing Service from a Remote Location.

Publish-subscribe monitoring is done by subscribing to the monitoring topics. See Chapter 6: Publish-Subscribe Monitoring of Queuing Service from a Remote Location.

To enable Request/Reply monitoring and configure its behavior, use the `<administration>` tag under `<queuing_service>` (See Configuring Administration (Section 3.3.3)).

To enable Pub/Sub monitoring and configure its behavior, use the `<monitoring>` tag under `<queuing_service>` (See Configuring Publish-Subscribe Monitoring (Section 3.3.4.2)).

By default, both, remote publish-subscribe monitoring and request-reply monitoring are turned off in Queuing Service for security and performance reasons. A `<monitoring>` or `<administration>` section is not required in the configuration file.

For example:

```xml
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="rti_queuing_service.xsd">
<queuing_service name="QueuingService_1">
  <administration>
    <domain_id>55</domain_id>
  </administration>
  <monitoring>
    <domain_id>55</domain_id>
  </monitoring>
  ...
</queuing_service>
</dds>
```

There are two kinds of monitoring data for an entity (for example, a SharedReaderQueue):

- Entity data
- Entity status

Entity data provides information about the configuration of the entity. For example, the service data contains a list of the SharedReaderQueues contained in the service. Entity data information is updated every time there is a configuration change that affects that data.

Entity status provides information about the operational status of an entity. This kind of information changes continuously and is computed and published periodically. For example, the SharedReaderQueue status contains information such as the SharedReaderQueue’s latency and throughput.

The following table shows the monitoring information available with publish-subscribe and request-reply monitoring:

<table>
<thead>
<tr>
<th></th>
<th>Publish-Subscribe</th>
<th>Request-Reply</th>
</tr>
</thead>
<tbody>
<tr>
<td>ServiceData</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SharedReaderQueueData</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>SharedReaderQueueStatus</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
For more information on how to retrieve the monitoring data, see Chapter 5: Administering Queuing Service from a Remote Location and Chapter 6: Publish-Subscribe Monitoring of Queuing Service from a Remote Location.

### 3.3.4.1 Configuring Request-Reply Monitoring

See Configuring Administration (Section 3.3.3).

### 3.3.4.2 Configuring Publish-Subscribe Monitoring

When publish-subscribe remote monitoring is enabled, Queuing Service will create one Domain-Participant, one Publisher, and one DataWriter to publish SharedReaderQueue status. You can configure the QoS of these entities with the `<monitoring>` tag defined under `<queuing_service>`.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;monitoring&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;enabled&gt;</code></td>
<td>Enables/disables publish-subscribe monitoring for the Queuing Service instance. Setting this value to true (default value) in the <code>&lt;monitoring&gt;</code> tag under <code>&lt;queuing_service&gt;</code> enables monitoring in all the entities unless they explicitly disable it by setting this tag to false in their local <code>&lt;entity_monitoring&gt;</code> tags. Setting this tag to false in the <code>&lt;monitoring&gt;</code> tag under <code>&lt;queuing_service&gt;</code> disables monitoring in all the Queuing Service entities. In this case, any monitoring configuration settings in the entities are ignored. Default value: true</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;datawriter_qos&gt;</code></td>
<td>Configures the DataWriter QoS for remote monitoring. If the tag is not defined, Queuing Service will use the Connext DDS defaults with this change: durability.kind = DDS_TRANSIENT_LOCAL_DURABILITY_QOS</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;domain_id&gt;</code></td>
<td>Specifies which domain ID Queuing Service will use to enable remote monitoring.</td>
<td>1 (required)</td>
</tr>
<tr>
<td><code>&lt;participant_qos&gt;</code></td>
<td>Configures the DomainParticipant QoS for remote monitoring. If the tag is not defined, Queuing Service will use the Connext DDS defaults with this change: resource_limits.type_code_max_serialized_length = 4096</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
The `<status_publication_period>` defined under `<queuing_service>/<monitoring>` is inherited by all the monitorable entities. An entity can override this value using the `<entity_monitoring>` tag.

For example, this how a SharedReaderQueue would override the status publication period:

```xml
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="rti_queuing_service.xsd">
  <queuing_service name="QueuingService_1">
    ...
    <monitoring>
      <domain_id>55</domain_id>
      <status_publication_period>
        <sec>5</sec>
        <nanosec>0</nanosec>
      </status_publication_period>
    </monitoring>
    ...  
    <shared_subscriber name="SharedSubscriber_1">
      ...
      <shared_reader_queue
        name="SharedReaderQueue_1"
        session="Session_1">
        ...
        <entity_monitoring>
          <enabled>true</enabled>
          <status_publication_period>
            <sec>3</sec>
            <nanosec>0</nanosec>
          </status_publication_period>
          ...
        </entity_monitoring>
      </shared_reader_queue>
    </shared_subscriber>
  </queuing_service>
</dds>
```

## Table 3.5 Monitoring Tags

<table>
<thead>
<tr>
<th>Tags within &lt;monitoring&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;publisher_qos&gt;</td>
<td>Configures the Publisher QoS for remote monitoring.</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>If the tag is not defined, Queuing Service will use the Connext DDS defaults.</td>
<td></td>
</tr>
<tr>
<td>&lt;status_publication_period&gt;</td>
<td>Specifies the frequency at which the status of an entity is published.</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;status_publication_period&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;sec&gt;3&lt;/sec&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;nanosec&gt;0&lt;/nanosec&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt;/status_publication_period&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the tag is not defined, the period is 5 seconds.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The status publication period defined in <code>&lt;queuing_service&gt;</code>/</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;monitoring&gt;</code> is inherited by all the monitorable entities within <code>&lt;queuing_service&gt;</code>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>An entity can override the period.</td>
<td></td>
</tr>
</tbody>
</table>

### 3.3.4.2.1 Publish-Subscribe Monitoring Configuration Inheritance

The `<status_publication_period>` defined under `<queuing_service>`/`<monitoring>` is inherited by all the monitorable entities. An entity can override this value using the `<entity_monitoring>` tag.

For example, this how a SharedReaderQueue would override the status publication period:
In the above example, the SharedReaderQueue overrides the status publication period, setting it to 3 seconds.

### 3.3.4.3 Configuring Statistics Calculation Process

*Queueing Service* reports multiple statistics as part of the entity status. Some of these statistics are counters, such as the number of samples received by a SharedReaderQueue; other statistics are statistics variables, such as the number of samples enqueued per second in a SharedReaderQueue.

```c
struct SharedReaderQueueStatus {
    ...
    unsigned long long received_message_count;
    ...
    StatisticVariable enqueue_throughput;
    ...
};
```

For a given statistic variable, *Queueing Service* computes the metrics in *StatisticMetrics* during specified time frames.

```c
struct StatisticMetrics {
    unsigned long long period_ms;
    long long count;
    float mean;
    float minimum;
    float maximum;
    float std_dev;
};
```

```c
struct StatisticVariable {
    StatisticMetric publication_period_metrics;
    sequence<StatisticMetrics, MAX_HISTORICAL_METRICS> historical_metrics;
};
```

The count is the sum of all values received during the time frame. For example, in the case of `enqueue_throughput`, count is the number of samples enqueued during the time frame.

---

<table>
<thead>
<tr>
<th>Tags within &lt;entity_monitoring&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;enabled&gt;</td>
<td>Enables/disables remote publish-subscribe monitoring for a given entity. If general monitoring is disabled, this value is ignored. Default value: true.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| <status_publication_period>    | Specifies the frequency at which the status of an entity is published. For example:  
```xml
<status_publication_period>
  <sec>3</sec>
  <nanosec>0</nanosec>
</status_publication_period>
```  
If the tag is not defined, its value is inherited from the general monitoring settings. | 0 or 1 |
XML Tags for Configuring Queuing Service

Queuing Service always calculates the statistics corresponding to the time between two status publications (publication_period_metrics field) independently of whether or not publish-subscribe monitoring is enabled. This time is configured using the tag <status_publication_period> under <monitoring> or <entity_monitoring> (Configuring Publish-Subscribe Monitoring (Section 3.3.4.2)).

You can also select additional windows on a per-entity basis using the tag <historical_statistics> under <statistics> (see Statistics Calculation (Section 3.3.4.3.1)). The sequence historical_metrics in StatisticVariable contains values corresponding to the windows that have been enabled:

- 5-sec. metrics correspond to activity in the last five seconds.
- 1-min. metrics correspond to activity in the last minute.
- 5-min. metrics correspond to activity in the last five minutes.
- 1-hour metrics correspond to activity in the last hour.
- Up-time metrics correspond to activity since the entity was created.

Each window has a field called period_ms that identifies its size in milliseconds. For the publication_period_metrics, this field contains the <status_publication_period>. For the up-time metrics, this field contains the time since the entity was created. For the other windows, this field contains a fixed value that identifies the window size (5000 for the 5-second window, 60000 for the one-minute window, etc).

3.3.4.3.1 Statistics Calculation

The accuracy of the statistics calculation process is determined by the value of the statistics sampling period. This period specifies how often statistics are gathered and is configured on a per entity basis using the tag <statistics_sampling_period> under <statistics>.

As a general rule, the statistics_sampling_period of an entity must be smaller than its status_publication_period for publish-subscribe monitoring and the request period for request-reply monitoring. A small statistics_sampling_period provides more accurate statistics at expense of increasing the memory consumption and decreasing performance.

The statistics calculation process is configured using the tags <statistics> under <queuing_service> and <shared_reader_queue>.

3.3.5 Configuring Persistence Settings

The <persistence_settings> tag configures the store settings that are used to persist the service state and the SharedReaderQueues's states (see Queuing Service Persistency (Section 2.14)). Table 3.8, Persistence Setting Tags lists the tags that you can specify in <persistence_settings>.

3.3.6 Configuring DomainParticipants

For each <domain_participant> tag, Queuing Service creates one DomainParticipant to communicate over DDS. Table 3.10, DomainParticipant Tags lists the tags allowed within <domain_participant>.
Table 3.7  **Statistics Tags**

<table>
<thead>
<tr>
<th>Tags within <code>&lt;statistics&gt;</code></th>
<th>Description</th>
<th>Number of Allowed Tags</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;historical_statistics&gt;</code></td>
<td>Enables or disables the statistic calculation within fixed time windows.</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>By default, Queuing Service only publishes the statistics corresponding to the window between two status publications. By using this tag, you can get the following additional windows:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 5 seconds</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 1 minute</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 5 minutes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• 1 hour</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Up time (since the entity was enabled)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>For example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;historical_statistics&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;five_second&gt;true&lt;/five_second&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;one_minute&gt;true&lt;/one_minute&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;five_minute&gt;false&lt;/five_minute&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;one_hour&gt;true&lt;/one_hour&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;up_time&gt;false&lt;/up_time&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/historical_statistics&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If a window is not present (inside <code>&lt;historical_statistics&gt;</code>), it is considered disabled.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Historical statistics can be overridden on a per entity basis.</td>
<td></td>
</tr>
<tr>
<td><code>&lt;statistics_sampling_period&gt;</code></td>
<td>Specifies the frequency at which statistics variables (such as throughput and latency) are updated.</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>For example:</td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;statistics_sampling_period&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;sec&gt;1&lt;/sec&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;nanosec&gt;0&lt;/nanosec&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;/statistics_sampling_period&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>If the tag is not defined, the period is 1 second.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>The statistic sampling period defined in <code>&lt;queuing_service&gt;</code> is inherited by all the entities inside <code>&lt;queuing_service&gt;</code>.</td>
<td></td>
</tr>
<tr>
<td></td>
<td>An entity can override the period.</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.8  **Persistence Setting Tags**

<table>
<thead>
<tr>
<th>Tags within <code>&lt;persistence_settings&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;filesystem&gt;</code></td>
<td>Configures the file system settings used to persist the service state and the SharedReaderQueues' states. See Table 3.9, Filesystem Tags.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
### Table 3.9 FilesysteTags

<table>
<thead>
<tr>
<th>Tags within &lt;filesystem&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;directory&gt;</td>
<td>Specifies the directory of the files in which the service state and the SharedReaderQueues’ states will be persisted. This directory can also be provided by, and is overridden by, the -persistentStoragePath command-line option. The directory must exist; otherwise the service will report an error upon start up. Default: Value provided with -persistentStoragePath or the current working directory if the command-line option is not provided.</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;file_max_size&gt;</td>
<td>This tag configures the maximum size (in KB) of the files storing the SharedReaderQueue data (both metadata and user data). Queuing Service will create a new file when this size is exceeded. Default: 1000 KB Note: This tag only applies to PERSISTENT SharedReaderQueues created with &lt;in_memory_state&gt; set to true (see Configuring SharedReaderQueues (Section 3.3.10)).</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;file_prefix&gt;</td>
<td>Specifies a name prefix associated with all the files created by Queuing Service. This prefix can also be provided by, and is overridden by, the -persistentFilePrefix command-line option. Default: Value provided with -persistentFilePrefix or 'QS' if the command-line option is not provided</td>
<td>0 or 1</td>
</tr>
<tr>
<td>&lt;journal_mode&gt;</td>
<td>Sets the journal mode of the persistent storage. This tag can take these values: • DELETE: Deletes the rollback journal at the conclusion of each transaction. • MEMORY: Stores the rollback journal in volatile RAM. This saves disk I/O. • OFF: Completely disables the rollback journal. If the application crashes in the middle of a transaction when the journal mode is set to OFF, the files containing the samples will very likely be corrupted. • PERSIST: Prevents the rollback journal from being deleted at the end of each transaction. Instead, the header of the journal is over-written with zeros. • TRUNCATE: Commits transactions by truncating the rollback journal to zero-length instead of deleting it. • WAL: Uses a write-ahead log instead of a rollback journal to implement transactions. Default: DELETE Note: This does not apply to PERSISTENT SharedReaderQueues created with &lt;in_memory_state&gt; set to true (see Configuring SharedReaderQueues (Section 3.3.10)).</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
XML Tags for Configuring Queuing Service

### Table 3.9  Filesystem Tags

<table>
<thead>
<tr>
<th>Tags within &lt;filesystem&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
</table>
| <synchronization>       | Determines the level of synchronization with the physical disk. This tag can take three values:  
  - FULL: Every sample is written to physical disk as Queuing Service receives it.  
  - NORMAL: Samples are written to disk at critical moments.  
  - OFF: No synchronization is enforced. Data will be written to physical disk when the OS flushes its buffers.  
  Default: OFF | 0 or 1 |
| <trace_file>             | Specifies the name of the trace file for debugging purposes. The trace file contains information about all SQL statements executed by the persistence service.  
  Default: No trace file is generated  
  Note: This does not apply to PERSISTENT SharedReaderQueues created with <in_memory_state> set to true (see Configuring SharedReaderQueues (Section 3.3.10)). | 0 or 1 |
| <vacuum>                 | Sets the auto-vacuum status of the storage. This tag can take these values:  
  - NONE: When data is deleted from the storage files, the files remain the same size.  
  - FULL: The storage files are compacted every transaction.  
  Default: FULL  
  Note: This does not apply to PERSISTENT SharedReaderQueues created with <in_memory_state> set to true (see Configuring SharedReaderQueues (Section 3.3.10)). | 0 or 1 |

### Table 3.10  DomainParticipant Tags

<table>
<thead>
<tr>
<th>Tags within &lt;domain_participant&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;domain_id&gt;</td>
<td>Specifies the domain ID associated with the DomainParticipant</td>
<td>1 (required)</td>
</tr>
</tbody>
</table>
| <memory_management>             | Controls how to allocate the memory for a sample buffer.  
  For details, see Configuring Memory Management for Sample Buffers (Section 3.3.6.1). | 0 or 1 |
| <participant_qos>               | Configures the DomainParticipant QoS. If the tag is not defined, Queuing Service will use the Connext DDS defaults. | 0 or 1 |
| <shared_subscriber>             | Configures a SharedSubscriber. See Configuring SharedSubscribers (Section 3.3.7). | 1 or more (required) |
3.3.6.1 Configuring Memory Management for Sample Buffers

For every sample in a SharedReaderQueue, Queuing Service uses a buffer to store the content of the sample in serialized form with CDR representation. The `<memory_management>` tag controls how to allocate the memory for a sample buffer.

For example:

```xml
<memory_management>
  <sample_buffer_min_size>16000</sample_buffer_min_size>
  <sample_buffer_trim_to_size>true</sample_buffer_trim_to_size>
</memory_management>
```

- **sample_buffer_min_size**: If the serialized size of an incoming sample is smaller or equal to this value, Queuing Service will use a pre-allocated buffer (with size equal to this value) from a pool to hold the sample.

  The initial and maximum number of buffers in the pool as well as the pool’s growth policy is configured using the XML tag `<resource_limits>` under `<shared_reader_queue>`/`<queue_qos>`.

  When the serialized size of the incoming sample is greater than this value, Queuing Service will allocate the buffer from the heap dynamically upon sample reception.

- **sample_buffer_trim_to_size**: This value controls what to do with the buffers that are dynamically allocated. When true, the buffers will be released when the corresponding samples are remove from the SharedReaderQueues. When false, the buffers are kept around for future samples. They maybe released later on but only to be replaced by bigger buffers.

Ranges:

- **sample_buffer_min_size**: -1 (In a SharedReaderQueue is the maximum serialized size of its samples) or positive number.

- **sample_buffer_trim_to_size**: true or false

Defaults:

- **sample_buffer_min_size**: 256
- **sample_buffer_trim_to_size**: true

Notice that setting a positive value for **sample_buffer_min_size** is critical when a data type has a very high maximum serialized size (e.g., megabytes) but most of the samples sent are much smaller than the maximum possible size (e.g., kilobytes). In this case, the memory footprint is reduced dramatically, while still correctly handling the rare cases in which very large samples are published.

3.3.7 Configuring SharedSubscribers

SharedSubscribers are containers that host SharedReaderQueues, allowing remote QueueConsumers to attach to the shared queues and providing “exactly once” or “at-most once” access to the samples in the shared queues.

With these access modes, when one QueueConsumer gets a message, the other QueueConsumers attached to the same SharedReaderQueue do not get that message. A SharedSubscriber can host one or more SharedReaderQueues, each one associated with a different DDS Topic name.

Table 3.11 lists the tags allowed within `<shared_subscriber>`.
### 3.3.8 Configuring Session Settings

Table 3.12, Session Settings Tags lists the only tag allowed within `<session_settings>`.

#### Table 3.12  Session Settings Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;session_settings&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;session&gt;</code></td>
<td>A session defines a threaded context for a SharedReaderQueue. See Configuring SharedSubscribers Sessions (Section 3.3.9)</td>
<td>1 or more (required)</td>
</tr>
</tbody>
</table>

### 3.3.9 Configuring SharedSubscribers Sessions

A session defines a threaded context for a SharedReaderQueue. SharedReaderQueues in different sessions can be processed in parallel. Sessions are part of SharedSubscribers.

For each Session defined within the tag `<session_settings>`, Queuing Service will create the following elements:

- Two threads: one for storing samples into SharedReaderQueues, and one to distribute samples from the SharedReaderQueues to QueueConsumers.
- One DDS Publisher
- One DDS Subscriber

The QoS of the Publisher and Subscriber are configured using the tags `<publisher_qos>` and `<subscriber_qos>` under `<shared_subscriber>`.

Table 3.13, Session Tags lists the tags allowed within `<session>`.
SharedReaderQueues and DeadLetterSharedReaderQueues can be associated with a session by using the XML attribute `session` in `<shared_reader_queue>` and `<dead_letter_shared_reader_queue>`, respectively.

### Table 3.13 Session Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;session&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;dequeue_period&gt;</code></td>
<td>Configures the period at which Queuing Service retries sending samples that have not been delivered to a QueueConsumer upon reception. This can happen when the available QueueConsumers cannot accept the samples or if there are no QueueConsumers in the system for a SharedReaderQueue. Example: <code>&lt;session&gt;</code> <code>&lt;thread&gt;</code> <code>&lt;dequeue_period&gt;</code> <code>&lt;sec&gt;1&lt;/sec&gt;</code> <code>&lt;nanosec&gt;0&lt;/nanosec&gt;</code> <code>&lt;/dequeue_period&gt;</code> <code>&lt;/thread&gt;</code> <code>&lt;/session&gt;</code> Default: 10 msec</td>
<td></td>
</tr>
<tr>
<td><code>&lt;monitoring&gt;</code></td>
<td>Enables and configures remote Pub/Sub monitoring for the SharedReaderQueue. See Configuring Monitoring (Section 3.3.4) and Chapter 6: Publish-Subscribe Monitoring of Queuing Service from a Remote Location.</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;replication&gt;</code></td>
<td>Enables SharedReaderQueue replication. See Chapter 7: High Availability.</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;statistics&gt;</code></td>
<td>Configures the statistic gathering process for publish-subscribe or request-reply monitoring of the SharedReaderQueue. See Configuring Monitoring (Section 3.3.4).</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
| `<thread>`             | Sets the mask, priority and stack size of the threads associated with this session. Example: `<session>` `<thread>` `<mask>MASK_DEFAULT</mask>` `<priority>` THREAD_PRIORITY_DEFAULT `</priority>` `<stack_size>` THREAD_STACK_SIZE_DEFAULT `</stack_size>` `</thread>` `</session>` Defaults:  
  - `mask = MASK_DEFAULT`
  - `priority = THREAD_PRIORITY_DEFAULT`
  - `stack_size = THREAD_STACK_SIZE_DEFAULT` | 0 or 1 |
### 3.3.10 Configuring SharedReaderQueues

A SharedReaderQueue is a logical DataReader queue hosted inside a SharedSubscriber that provides “exactly once” or “at-most once” access to the Consumers attached to the SharedReaderQueue. It is associated with a Topic and the name of the SharedReaderQueue is derived from the name of the Topic and the SharedSubscriber that hosts it. Implementation-wise, a SharedReaderQueue is composed of an input (DDS DataReader) and output (DDS DataWriter) pair that, together with a queue storage, implement the queuing behavior for a Topic.

The input DataReader is matched to the DataWriters associated with the QueueProducers and the output DataWriter is matched to the DataReaders associated with the QueueConsumers. The processing logic ensures that each sample in the SharedReaderQueue is delivered to only one of the QueueConsumers.

- Table 3.14, SharedReaderQueue Tags and Table 3.15, Queue QoS Tags describe the tags allowed within `<shared_reader_queue>`.
- Table 3.16, `<shared_reader_queue>` Attributes describes the attributes you may set for `<shared_reader_queue>`.

#### Table 3.14 SharedReaderQueue Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;shared_reader_queue&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
</table>
| `<datareader_qos>`                | Configures the QoS for the SharedReaderQueue DataReader. If the tag is not defined, Queuing Service will use the Connext DDS defaults with the following changes:  
  - `reliability.kind = DDS_RELIABLE_RELIABILITY_QOS`  
  - `reliability.acknowledgment_kind = APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE`  
  - `history.kind = DDS_KEEP_ALL_HISTORY_QOS`  
  - `reader_resource_limits.max_app_ack_response_length = 1`  
  - `subscription_name.role_name = QUEUING_SERVICE`  
  - `service.kind = QUEUING_SERVICE_QOS`  
  (this value cannot be changed) | 0 or 1 |
| `<datawriter_qos>`                | Configures the QoS for the SharedReaderQueue DataWriter. If the tag is not defined, Queuing Service will use the Connext DDS defaults with the following changes:  
  - `reliability.kind = DDS_RELIABLE_RELIABILITY_QOS`  
  - `reliability.acknowledgment_kind = APPLICATION_EXPLICIT_ACKNOWLEDGMENT_MODE`  
  - `history.kind = DDS_KEEP_ALL_HISTORY_QOS`  
  - `service.kind = QUEUING_SERVICE_QOS`  
  (this value cannot be changed) | 0 or 1 |
| `<queue_qos>`                     | Configures the QoS for the SharedReaderQueue. See Table 3.15, Queue QoS Tags. | 0 or 1 |
### SharedReaderQueue Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;shared_reader_queue&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;reply_topic&gt;</code></td>
<td>The topic name for the implicit Reply SharedReaderQueue created by setting <code>&lt;reply_type&gt;</code>. This tag is ignored if <code>&lt;reply_type&gt;</code> is not set. Default: <code>&lt;topic_name&gt;Reply</code></td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;reply_type&gt;</code></td>
<td>The name of the type associated with a Reply SharedReaderQueue. When it comes to the creation of a Reply SharedReaderQueue, you have two options: • Declare the queue explicitly in the configuration file. • Declare the queue implicitly through the usage of <code>&lt;reply_type&gt;</code>. In this case, the configuration of the Reply SharedReaderQueue matches the configuration of the SharedReaderQueue containing <code>&lt;reply_type&gt;</code>. See Sending a Reply from QueueConsumer to QueueProducer (Section 2.11).</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;topic_name&gt;</code></td>
<td>The name of the Topic associated with the SharedReaderQueue. QueueProducers will publish on this Topic. QueueConsumers will subscribe to a Topic with name “&lt;topic_name&gt;@SharedSubscriberName” where SharedSubscriberName is the name of the SharedSubscriber containing the SharedReaderQueue. See Queuing Service Entities (Section 2.6).</td>
<td>1 (required)</td>
</tr>
<tr>
<td><code>&lt;type_name&gt;</code></td>
<td>The name of the type associated with the ShareReaderQueue. See Configuring Queuing Service Types (Section 3.3.1).</td>
<td>1 (required)</td>
</tr>
<tr>
<td><code>&lt;update_datareader_qos&gt;</code></td>
<td>Configures the QoS of the DataReader used to receive the status information required by the SharedReaderQueue replication protocol. If the tag is not defined, Queuing Service will use the Connext defaults with the following changes: • <code>reliability.kind</code> = DDS_RELIABLE_RELIABILITY_QOS (this value cannot be changed) • <code>history.kind</code> = DDS_KEEP_ALL_HISTORY_QOS (this value cannot be changed)</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;update_datawriter_qos&gt;</code></td>
<td>Configures the QoS of the DataWriter used to publish the status information required by the SharedReaderQueue replication protocol. If the tag is not defined, Queuing Service will use the Connext defaults with the following changes: • <code>reliability.kind</code> = DDS_RELIABLE_RELIABILITY_QOS (this value cannot be changed) • <code>history.kind</code> = DDS_KEEP_ALL_HISTORY_QOS (this value cannot be changed)</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>
### Table 3.15  Queue QoS Tags

<table>
<thead>
<tr>
<th>Tags within &lt;queue_qos&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
</table>
| **<distribution>**     | Configures the dispatch policy for the SharedReaderQueue. Queuing Service uses the dispatch policy to determine which QueueConsumer gets each sample. In this release, Queuing Service only supports a ROUND_ROBIN dispatch policy, with and without explicit availability feedback from QueueConsumers. You can also configure a SharedReaderQueue so that the last sample in the SharedReaderQueue for a QueueConsumer is marked with the flag DDS_LAST_SHARED_READER_QUEUE_SAMPLE before is sent to the QueueConsumer. The QueueConsumer application can inspect the value of this flag by checking the flag field in SampleInfo. Example:  
```
<distribution>
  <kind>ROUND_ROBIN</kind>
  <mark_last_undelivered_sample>true</mark_last_undelivered_sample>
  <property>
    <value>
      <element>
        <name>UNACKED_THRESHOLD</name>
        <value>-1</value>
      </element>
    </value>
    <value>
      <element>
        <name>ALLOW_CONSUMER_FEEDBACK</name>
        <value>1</value>
      </element>
    </value>
  </property>
</distribution>
```
See Selecting a QueueConsumer for a Sample (Section 2.10) for more information regarding the dispatch policy. Defaults:  
- kind: ROUND_ROBIN  
- UNACKED_THRESHOLD: -1  
- ALLOW_CONSUMER_FEEDBACK: 0  
- mark_last_undelivered_sample: false | 0 or 1 |

| **<lifespan>**          | Configures how long a sample written by a QueueProducer is kept in the SharedReaderQueue. Example:  
```
<lifespan>
  <duration>
    <sec>60</sec>
    <nanosec>0</nanosec>
  </duration>
</lifespan>
```
Note: A finite lifespan set on the QueueProducer's DataWriter using the Lifespan QoS policy takes precedence over this value. Default: UNLIMITED (no lifespan) | 0 or 1 |
**XML Tags for Configuring Queuing Service**

### Table 3.15  Queue QoS Tags

<table>
<thead>
<tr>
<th>Tags within &lt;queue_qos&gt;</th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
</table>
| <persistence>           | Configures whether or not the SharedReaderQueue state must be persisted on disk for fault tolerance purposes:  
|                         | There are two values for this policy:  
|                         | • VOLATILE: Keep the samples in-memory.  
|                         | • PERSISTENT: Store the samples into disk.  
|                         | Example:  
|                         |   <persistence>  
|                         |     <kind>PERSISTENT</kind>  
|                         |   </persistence>  
|                         | In the case of PERSISTENT SharedReaderQueues, you can choose between two implementations using the XML tag <in_memory_state>:  
|                         | • Without In-Memory State: The metadata and user data associated with the SharedReaderQueue's samples is kept only on disk.  
|                         | • With In-Memory State: The metadata for the SharedReaderQueue's samples is always kept both on disk and in memory. The sample's user data is kept in memory and on disk only when:  
|                         | o Its serialized size is smaller than the threshold set with <domain_participant>/<memory_management>/<sample_buffer_min_size> (see Memory Management for a Sample (Section 2.15.2)).  
|                         | o <domain_participant>/<memory_management>/<sample_buffer_trim_to_size> is false (see Memory Management for a Sample (Section 2.15.2)).  
|                         | Example:  
|                         |   <persistence>  
|                         |     <kind>PERSISTENT</kind>  
|                         |     <in_memory_state>true</in_memory_state>  
|                         |   </persistence>  
|                         | Default:  
|                         | • kind: VOLATILE  
|                         | • in_memory_state: true  
|                         | See Queuing Service Persistency (Section 2.14). | 0 or 1 |
### XML Tags for Configuring Queuing Service

<table>
<thead>
<tr>
<th>Tags within <code>&lt;queue_qos&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
</table>
| `<redelivery>` | Configures the redelivery policy for the SharedReaderQueue. Example:  
```
<redelivery>
   <max_delivery_retries>
      10
   </max_delivery_retries>
   <response_timeout>
      <duration>
         <sec>60</sec>
         <nanosec>0</nanosec>
      </duration>
   </response_timeout>
</redelivery>
```
In the above example:  
- `<max_delivery_retries>` configures the maximum number of redelivery attempts for a sample in a SharedReaderQueue  
- `<response_timeout>` configures the maximum time that Queue Service waits for an acknowledgment from the QueueConsumer to which the sample was sent to. After that timeout expires, the sample is redelivered to a different QueueConsumer for `<max_delivery_retries>`  
Defaults:  
- For `<max_delivery_retries>`: 0  
- For `<response_timeout>`: UNLIMITED | 0 or 1 |
| `<reliability>` | Configures the QoS for reliable delivery of samples from a QueueProducer to the Queuing Service. This release supports only one configuration parameter, which allows you to disable the sending of application-level acknowledgement messages from Queue Service to the QueueProducers after samples are stored into a SharedReaderQueue. Example:  
```
<reliability>
   <app_ack_sample_to_producer>
      false
   </app_ack_sample_to_producer>
</reliability>
```
Default: `<app_ack_sample_to_producer> = true` | 0 or 1 |
| `<resource_limits>` | This policy:  
- Provides fine-grained control over the resources (memory and disk) associated with the samples in a SharedReaderQueue.  
- Provides a way to configure the behavior of a SharedReaderQueue when a new sample arrives and the SharedReaderQueue is full.  
- Provides ways to monitor when the space taken by the samples in a SharedReaderQueue goes above or below configurable watermarks and when the SharedReaderQueue fills up. For default values and additional information, see SharedReaderQueue Resource Management (Section 2.15). | 0 or 1 |
3.3.11 Configuring DeadLetterSharedReaderQueues

Queuing Service provides support for deal-letter queues. A deal-letter queue is a SharedReader-Queue to which other SharedReaderQueues can send messages that for some reason could not be successfully delivered and processed.

Queuing Service supports the definition of one deal-letter queue per SharedSubscriber by using the XML tag `<dead_letter_shared_reader_queue>`. The deal-letter queue has two limitations compared with a regular queue:

- It cannot have a `<reply_type>`.
- It cannot have a `<type_name>`.

By default, SharedReaderQueues do not send undelivered samples to the deal-letter queue. To enable this behavior, you must use the attribute `dead_letter_queue` in `<shared_reader_queue>`. This attribute must be set to the name of the deal-letter queue in the configuration file.

For more information, see Dead-Letter Queues (Section 2.12).
3.4 Using Variables in XML

The text within an XML tag can refer to a variable. To do so, use the following notation:

\[ $(MY_VARIABLE) \]

For example:

```xml
<element>
  <name>The name is $(MY_NAME)</name>
  <value>The value is $(MY_VALUE)</value>
</element>
```

When the XML parser parses the above tags, it will replace the references to variables with their actual values as follows:

1. First, it will try to get the variable value from the command-line. The variable value can be provided using the `-var` command-line option (see Queuing Service runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin. There are four ways to start Queuing Service: (Section )).

2. If the value is not found, the parser will try to get it from the OS environment variables.

3. If the value still cannot be found, the parsing process will fail.

3.5 Enabling RTI Distributed Logger in Queuing Service

Queuing Service provides integrated support for RTI Distributed Logger. When you enable Distributed Logger, Queuing Service will publish its log messages to Connext DDS.

You can use RTI Monitor to visualize the log message data. Since the data is provided in a topic, you can also use RTI DDS Spy (`rtiddsspy`) or even write your own visualization tool.

To enable Distributed Logger, modify the Queuing Service XML configuration file. In the `<administration>` section, add the `<distributed_logger>` tag as seen here:

```xml
<dds xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="rti_queuing_service.xsd">
  
  <queuing_service name="QueuingService_1">
    <administration>...
      <distributed_logger>
        <enabled>true</enabled>
      </distributed_logger>
    </administration>
  </queuing_service>
</dds>
```

There are more configuration tags that you can use to control Distributed Logger’s behavior. For example, you can specify a filter so that only certain types of log messages are published. For details, see the RTI Distributed Logger Getting Started Guide.
Chapter 4  Running Queuing Service

Queuing Service runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin`. There are four ways to start Queuing Service:

- Starting from Launcher  (Section 4.1)
- Starting Manually from the Command Line (Section 4.2)
- Using Queuing Service as a Windows Service (Section 4.3)

If you are starting Queuing Service as a Windows Service, also read Notes when Running as a Windows Service (Section 4.3.3).

4.1  Starting from Launcher

1. Start RTI Launcher from the Start menu (on Windows systems) or on the command line, type:
   
   `<NDDSHOME>/bin/rtilauncher`

2. From the Services tab, select Queuing Service:
4.2 Starting Manually from the Command Line

To start Queuing Service, enter:

```
cd <NDDSHOME>
bin/rtiqueuingservice [options]
```

Example:

```
cd <NDDSHOME>
bin/rtiqueuingservice -cfgFile example.xml -cfgName QueuingService_1
```

Table 4.1 describes the command-line options.

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-appName &lt;name&gt;</td>
<td>Assigns a name to the execution of Queuing Service. Remote commands will refer to the queuing service using this name. In addition, the name of DomainParticipants created by Queuing Service will be based on this name. Default: The name given with -cfgName, if present, otherwise it is RTI_Queuing_Service.</td>
</tr>
<tr>
<td>-cfgFile &lt;name&gt;</td>
<td>Specifies a configuration file to be loaded. This parameter is required. See Section 3.1, How to Load the XML Configuration, in the Queuing Service User's Manual.</td>
</tr>
<tr>
<td>-cfgName &lt;name&gt;</td>
<td>Specifies a configuration name. Queuing Service will look for a matching &lt;queueing_service&gt; tag in the configuration file.</td>
</tr>
<tr>
<td>-cfgRemote</td>
<td>Specifies that the initial configuration of the service must be obtained remotely from other running instances. Using this option also requires the use of -remoteAdministrationDomainId to enable remote administration, because the initial configuration will be received in the remote administration domain ID. If you use this option and -cfgName, the service will wait until a configuration with that name is received. Otherwise, the service will use the first configuration that it receives. If the service does not receive the initial configuration after a configurable timeout (see -cfgRemoteTimeout), it will load the configuration from the input configuration file(s).</td>
</tr>
<tr>
<td>-cfgRemoteTimeout &lt;n&gt;</td>
<td>Specifies the maximum amount of time, in seconds, that Queuing Service will wait for an initial configuration when using -cfgRemote. Default: 20 seconds</td>
</tr>
<tr>
<td>-daemon</td>
<td>Runs Queuing Service as a daemon/Windows service. When this flag is present, Queuing Service will start in the background. Note that some systems may require special privileges to do this.</td>
</tr>
<tr>
<td>-domainIdBase &lt;ID&gt;</td>
<td>Sets the base domain ID. This value is added to the domain IDs in the configuration file. For example, if you set –domainIdBase to 50 and use domainIDs 0 and 1 in the configuration file, then Queuing Service will use domains 50 and 51. Default: 0</td>
</tr>
</tbody>
</table>
Starting Manually from the Command Line

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-heapSnapshotPeriod</code></td>
<td>Enables heap monitoring. <em>Queueing Service</em> will generate a heap snapshot every <code>&lt;sec&gt;</code>. Default: heap monitoring is disabled.</td>
</tr>
<tr>
<td><code>-heapSnapshotDir</code></td>
<td>When heap monitoring is enabled, this parameter configures the directory where the snapshots will be stored. The snapshot filename format is <code>RTI_&lt;configurationName&gt;&lt;processId&gt;&lt;index&gt;.log</code>. Default: current working directory</td>
</tr>
<tr>
<td><code>-help</code></td>
<td>Displays help information.</td>
</tr>
<tr>
<td><code>-remoteAdministrationDomainId &lt;ID&gt;</code></td>
<td>Enables remote administration and sets the domain ID for remote communication. When remote administration is enabled, <em>Queueing Service</em> will create a <code>DomainParticipant</code>, <code>Publisher</code>, <code>Subscriber</code>, <code>DataWriter</code>, and <code>DataReader</code> in the designated domain. See Chapter 5, Administering <em>Queueing Service</em> from a Remote Location, in the <em>Queueing Service</em> User’s Manual. This option overrides the value of the tag <code>&lt;domain_id&gt;</code> within a <code>&lt;administration&gt;</code> tag. This parameter is required when using <code>-cfgRemote</code>. Default: Remote administration is not enabled unless it is enabled from the XML file.</td>
</tr>
<tr>
<td><code>-persistentFilePrefix</code></td>
<td>Specifies a name prefix to use with all files created by <em>Queueing Service</em>. This option overrides the value of the tag <code>&lt;file_prefix&gt;</code> within <code>&lt;persistence_settings&gt;/&lt;filesystem&gt;</code>. Default: Value in <code>&lt;persistence_settings&gt;/&lt;filesystem&gt;/&lt;file_prefix&gt;</code>.</td>
</tr>
<tr>
<td><code>-persistentStoragePath</code></td>
<td>Configures the directory for persistent storage. This option overrides the value of the tag <code>&lt;directory&gt;</code> within <code>&lt;persistence_settings&gt;/&lt;filesystem&gt;</code>. Default: Value in <code>&lt;persistence_settings&gt;/&lt;filesystem&gt;/&lt;directory&gt;</code>.</td>
</tr>
<tr>
<td><code>-var &lt;name&gt;=&lt;value&gt;</code></td>
<td>Sets the value of the variable <code>&lt;name&gt;</code>. This variable can be referenced within the XML configuration files using the <code>$&lt;name&gt;</code> notation. See Section 3.4, Using Variables in XML, in the <em>Queueing Service</em> User’s Manual for more information on configuration variables. You may have more than one <code>-var</code> flag on the command line. On Windows platforms, you will need to put quotation marks around the variable name and value, like this: <code>-var &quot;MY_VAR=myvalue&quot;</code></td>
</tr>
<tr>
<td><code>-verbosity &lt;n&gt;</code></td>
<td>Controls what type of messages are logged: 0 - Silent 1 - Exceptions (<em>Connext DDS</em> and <em>Queueing Service</em>) (default) 2 - Warnings (<em>Queueing Service</em>) 3 - Information (<em>Queueing Service</em>) 4 - Warnings (<em>Connext DDS</em> and <em>Queueing Service</em>) 5 - Tracing (<em>Queueing Service</em>) 6 - Tracing (<em>Connext DDS</em> and <em>Queueing Service</em>) Each verbosity level, <code>&lt;n&gt;</code>, includes all the verbosity levels smaller than <code>&lt;n&gt;</code>.</td>
</tr>
<tr>
<td><code>-version</code></td>
<td>Prints the <em>Queueing Service</em> version number.</td>
</tr>
</tbody>
</table>
4.3 Using Queuing Service as a Windows Service

Windows Services automatically run in the background when the system reboots.

4.3.1 Enabling Queuing Service to Run as a Windows Service

If you want to run Queuing Service as a Windows Service, you must install it as such before running it. To install it as a Windows Service, run the following command in a terminal with Administrator privileges:

```bash
<NDDSHOME>/bin/rtiqueuingservice -installService
```

By default, Queuing Service is installed with the service name `rtiqueuingservice523`. If you want to install it with a different service name, you can use the `-serviceName` flag. For instance (you would enter this all on one line):

```bash
<NDDSHOME>/bin/rtiqueuingservice -installService -serviceName mycustomservicename
```

Using the `-serviceName` parameter with different names allows you to install multiple Queuing Service instances on the same host.

4.3.2 Running RTI Queuing Service as a Windows Service

If you added Queuing Service as a Windows Service and want to run it without rebooting, you can start it as a service from the command line with the Windows `sc` utility:

```bash
sc <serviceName> start
```

By default, it will start Queuing Service with the "defaultService" configuration that is stored in `<NDDSHOME>/resource/xml/RTI_QUEUING_SERVICE.xml`. This configuration contains a service running with an empty SharedSubscriber with remote administration and monitoring enabled.

If you want to start Queuing Service with different parameters, you can use the utility `nssm`. You can specify the parameters from the command line by setting the option `AppParameters`. For example (you would enter this all on one line):

```bash
%NDDSHOME%/resource/app/bin/x64Win64VS2008/nssm.exe set <serviceName> AppParameters "<queuing service arguments>"
```

For more information and examples, see Notes when Running as a Windows Service (Section 4.3.3).

Additionally, you can start Queuing Service from the Windows Services Control Manager. From the Start Menu, select Control Panel, Administrative Services, Services. Click on the service in the list, then right-click to select Start.

4.3.3 Notes when Running as a Windows Service

Here are some things to consider when running Queuing Service as a Windows Service:

- All AppParameters arguments must be enclosed in quotation marks.
- To refer to variables in the XML configuration file, use the Queuing Service command-line option `-var` to set the variable’s value. The syntax for referring to a variable in the XML file is:

  ```xml
  <name>$(NAME)</name>
  ```
For the AppParameters passed to nssm, use -var like this:

    -var MY_DOMAIN=10

For example (you would enter this all on one line):

    %NDDSHOME%\resource\app\bin\x64Win64VS2008\nssm.exe set
    rtiqueuingservice523 AppParameters
    "-cfgFile search for a file with spaces'qsconf-with-vars.xml'
    -cfgName MyCustomConf -var MY_DOMAIN=10"

If a variable value includes spaces, you must enclose the value in escaped quotes. For example (you would enter this all on one line):

    %NDDSHOME%\resource\app\bin\x64Win64VS2008\nssm.exe
    set rtiqueuingservice523 AppParameters
    "-cfgFile search for a file with spaces'qsconf-with-vars.xml'
    -cfgName MyCustomConf -var "NAME=My QS name"

If you use environment variables instead of the -var command-line option, you may need to restart your Windows machine.

If you specify -cfgFile in the Start Parameters field, you must use the full path to the file.

Some versions of Windows do not allow Windows Services to communicate with other services/applications using shared memory. For this reason, if you plan to run Queuing Service as Windows Service, you should disable the shared-memory transport in all the DomainParticipants created by Queuing Service and in the applications communicating with Queuing Service. For more information on setting builtin transports, see the RTI Connext DDS Core Libraries User’s Manual (Section 15.1, Builtin Transport Plugins).

In some scenarios, you may need to add a multicast address (e.g., builtin.udpv4://239.255.0.1) to your discovery peers. For details on setting the discovery peers, see the RTI Connext DDS Core Libraries Getting Started Guide (Section 4.1.2, How to Set Your Discovery Peers).

### 4.3.4 Stopping Queuing Service when it is Running as a Windows Service

To stop Queuing Service when it is running as a Windows Service, use this command:

    sc rtiqueuingservice523 stop

You can also start/stop Queuing Service from the Windows Services Control Manager. From the Start menu, select Control Panel, Administrative Services, Services. Click on the service in the list, then right-click to select Start or Stop.

### 4.3.5 Disabling Queuing Service from Running as a Windows Service

To remove Queuing Service from the list of Windows Services on your system, run this command in a terminal with Administrator privileges:

    <NDDSHOME>\bin\rtiqueuingservice -uninstallService

By default, the service rtiqueuingservice523 is uninstalled. If you want to uninstall a different service instance, add the -serviceName option to the above command.
Chapter 5  Administering Queuing Service from a Remote Location

Queuing Service can be controlled remotely by sending commands through a special topic. Any Connext DDS application can be implemented to send these commands and receive the corresponding responses.

5.1 Enabling Remote Administration

By default, remote administration is disabled in Queuing Service for security reasons.

To enable remote administration, you can use the `<administration>` tag (see Configuring Administration (Section 3.3.3)) or the `-remoteAdministrationDomainId <ID>` command-line parameter, which enables remote administration and sets the domain ID for remote communication. For more information about the command-line options, see Queuing Service runs as a separate application. The script to run the executable is in `<NDDSHOME>/bin`. There are four ways to start Queuing Service: (Section ).

When remote administration is enabled, Queuing Service will create a DomainParticipant, Publisher, Subscriber, DataWriter, and DataReader in the designated domain. (The QoS values for these entities are described in Configuring Administration (Section 3.3.3).)

5.2 Remote Administration API

Queuing Service provides a RESTful-style remote administration API in which the commands have the following format:

```
<ACTION> <target_queuing_service> <resource_identifier> [body]
```

Where:

- `<ACTION>` is one the following values: CREATE, DELETE, GET, SHUTDOWN.
- `<target_queuing_service>` can be:
  - The application name of a Queuing Service instance, such as “MyQueuingService1”, as specified at start-up with the command-line option `-appName` (see Table 4.1, RTI Queuing Service Command-Line Options)
  - A regular expression—as defined by the POSIX fnmatch API (1003.2-1992 section B.6)—for a Queuing Service application name, such as “MyQueuingService*”
Remote Administration API

- `<resource_identifier>` identifies the resource to which the action is applied (see Resource Identifiers (Section 5.2.1)).
- `<body>` identifies the parameters of the action on the resource identified by `<resource_identifier>`. For example, when creating a SharedReaderQueue, the body is the XML snippet for the new queue.

### 5.2.1 Resource Identifiers

The format of a resource identifier is as follows:

```
/<resource_kind_1>/<resource_name_1>/.../<resource_kind_N>[/<resource_name_N>]
```

Where:

- `<resource_kind>` can have one of the following values:
  - `domain_participant`, `shared_subscriber`, `shared_reader_queue`, `dead_letter_shared_reader_queue`, `status`, `data`, and `message`. The resource kinds `status`, `data`, and `message` represent different information for an entity.
  - `status`: Refers to the operational status for a Queuing Service entity. This information changes continuously. The status information is composed primarily of statistics.
  - `data`: Refers to configuration data. This data is mostly static and does not change continuously.
  - `message`: Applies to SharedReaderQueues and refers to samples in the queues.

- `<resource_name>` specifies the name of the resource as defined in the XML configuration file using the attribute name.

For example, consider the following XML:

```xml
<?xml version="1.0"?>
<dds>
  <queuing_service name="QueuingService_1">
    ...
    <domain_participant name="DomainParticipant_1">
      ...
      <shared_subscriber name="SharedSubscriber_1">
        ...
        <shared_reader_queue name="SharedReaderQueue_1">
          </shared_reader_queue>
        </shared_subscriber>
      </domain_participant>
  </queuing_service>
</dds>
```

The resource identifier for the `DomainParticipant` is: `/domain_participant/DomainParticipant_1`.

The resource identifier for SharedSubscriber is: `/domain_participant/DomainParticipant_1/shared_subscriber/SharedSubscriber_1`.

The resource identifier for the SharedReaderQueue is: `/domain_participant/DomainParticipant_1/shared_subscriber/SharedSubscriber_1/shared_reader_queue/SharedReaderQueue_1`.

The resource identifier for the sample(s) in the SharedReaderQueue is: `/domain_participant/DomainParticipant_1/shared_subscriber/SharedSubscriber_1/shared_reader_queue/SharedReaderQueue_1/message`.
The resource identifier for the SharedReaderQueue status is: /domain_participant/
DomainParticipant_1/shared_subscriber/SharedSubscriber_1/shared_reader_queue/
SharedReaderQueue_1/status.

5.2.2 Sample Selector

For requests that apply to messages in a SharedReaderQueue, you may optionally provide a
sample selector as part of the <body>. The sample selector is an SQL-like expression.

Expression Grammar:

```
Condition ::= Predicate
  | Condition 'AND' Condition
  | Condition 'OR' Condition
  | 'NOT' Condition
  | '(' Condition ')' 

Predicate ::= ComparisonPredicate 

ComparisonPredicate ::= ComparisonTerm RelOp ComparisonTerm 

ComparisonTerm ::= FieldIdentifier | Parameter 

FieldIdentifier ::= FIELDNAME 

RelOp ::= '=' | '<>' 

Parameter ::= 
  | SEQUENCE_NUMBER |
  | INTEGER_VALUE|
  | BOOLEAN_VALUE|
  | STRING|
  | OCTET_ARRAY 
```

Token Expressions:

- **FIELDNAME**—A reference to a field in the data structure. A period `.` is used to navigate
  through nested structures. The number of dots that may be used in a FIELDNAME is
  unlimited. An '@' symbol prepending the field indicates that the field is a metadata field.

- **INTEGER_VALUE**—Any series of digits, optionally preceded by a plus or minus sign,
  representing a decimal integer value within the range of the system. 'L' or 'l' must be
  used for long long, otherwise long is assumed. A hexadecimal number is preceded by 0x
  and must be a valid hexadecimal expression.

- **BOOLEAN_VALUE**—Can either be TRUE or FALSE, and is case insensitive.

- **STRING**—Any series of characters encapsulated in single quotes, except the single quote
  itself.

- **OCTET_ARRAY**—An array of octets represented as follows: `&hex(hex_octet_values)`. For example:
  
  &hex(070809A0B0C0D0E0F10111213141516)

  Here the left-most pair represents the byte and index 0.

- **SEQUENCE_NUMBER**—A sequence number represented by a pair (high, low). For example:
  
  (2,3)
Supported Field Names:

The only field names supported in this release are:

- `@original_sample_identity.writer_guid.value`
- `@original_sample_identity.sequence_number`
- `@sample_queue_status`

The `original_sample_identity` identifies a sample sent by the QueueProducer. The identity consists of a pair (Virtual Writer GUID, Virtual Sequence Number).

By default, the identity of a sample published with a QueueProducer’s `DataWriter` is automatically set by the middleware. You can access this value by using the `write_w_params()` operation. It is also possible to explicitly set the sample identity by using the same `write_w_params()` operation. For details on how to set and retrieve the sample identity, see Section 6.3.8, Writing Data, in the RTI Connext DDS Core Libraries User’s Manual.

The `sample_queue_status` is a mask that represents the status of a sample in a SharedReader-Queue. The possible statuses are:

- UNDELIVERED_MESSAGE_STATUS
- SENT_MESSAGE_STATUS
- DELIVERED_MESSAGE_STATUS

Sample Selector Examples:

To select all the samples that have been sent to a QueueConsumer but not acknowledged yet:

```text
@sample_queue_status = SENT_MESSAGE_STATUS
```

To select all the samples that have been not been delivered to a QueueConsumer yet:

```text
@sample_queue_status = SENT_MESSAGE_STATUS | UNDELIVERED_MESSAGE_STATUS
```

To select all the samples coming from a QueueProducer’s `DataWriter` identified by virtual GUID 1:

```text
@original_sample_identity.writer_guid.value = 
&hex(00000000000000000000000000000001)
```

To select the sample coming from a QueueProducer’s `DataWriter` identified by virtual GUID 1 with sequence number 1:

```text
@original_sample_identity.writer_guid.value = 
&hex(00000000000000000000000000000001) AND 
@original_sample_identity.sequence_number = (0,1)
```

5.3 Remote Administration Topics

For remote administration, Queuing Service creates two topics:

- `rti/service/admin/command_request` is used to send a command from a client to Queuing Service.
- `rti/service/admin/command_reply` is used to send the command response(s) from Queuing Service to the client.

The topics have these corresponding types:

- `RTI::Service::Admin::CommandRequest`
5.4 Remote Commands in Queuing Service

This section describes the remote commands available in Queuing Service. Accessing Queuing Service from a Connext DDS application (Section 5.5) explains how to use remote administration from a Connext DDS application.

5.4.1 Create SharedReaderQueue

The following command is used to create a SharedReaderQueue:

```
CREATE <target_queuing_service> <shared_subscriber_resource_identifier> <xml_url>
```

Where:

- `<shared_subscriber_resource_identifier>` is the resource identifier for the SharedSubscriber that will contain the SharedReaderQueue.
- `<xml_url>` contains an XML snippet containing the SharedReaderQueue configuration. A full file (starting with `<dds>...`) is not valid. For example:

```
str://"<shared_reader_queue name=""SharedReaderQueue_1""...>
    <topic_name>RequestMessageTopic</topic_name> ...
</shared_reader_queue>"
```

Return Value:

Upon success, this command returns RTI_SERVICE_COMMAND_REPLY_OK in the `retcode` field of the reply. Otherwise, this command returns one of the following values, and the field `string_body` contains a human-readable string describing the error:

- RTI_SERVICE_COMMAND_REPLY_ALREADY_EXISTS: If the SharedReaderQueue already exists with a different configuration.
Remote Commands in Queuing Service

- RTI_SERVICE_COMMAND_REPLY_DUPLICATED: If the SharedReaderQueue already exists with the same configuration.
- RTI_SERVICE_COMMAND_REPLY_ERROR: For any other creation error.

5.4.2 Delete SharedReaderQueue

The following command is used to delete a SharedReaderQueue:

```
DELETE <target_queuing_service> <shared_reader_queue_resource_identifier>
```

Return Value:
Upon success, this command returns RTI_SERVICE_COMMAND_REPLY_OK in the `retcode` field of the reply. Otherwise, this command returns RTI_SERVICE_COMMAND_REPLY_ERROR, and the field `string_body` contains a human-readable string describing the error.

5.4.3 Flush SharedReaderQueue

The following command is used to flush all the samples or a set of samples from a SharedReaderQueue:

```
DELETE <target_queuing_service> <shared_reader_queue_resource_identifier> /message <sample selector>
```

Parameters:
The `<sample_selector>` (see Sample Selector (Section 5.2.2)) is a SQL expression that specifies the set of samples that must be removed and it must be provided in the field `string_body` of the CommandRequest.

Return Value:
Upon success, this command returns RTI_SERVICE_COMMAND_REPLY_OK in the `retcode` field of the reply. Otherwise, this command returns RTI_SERVICE_COMMAND_REPLY_ERROR, and the field `string_body` contains a human-readable string describing the error.

5.4.4 Get SharedReaderQueue Status

The type of the SharedReaderQueue’s status is called SharedReaderQueueStatus; you can find it in the file `<NDDSHOME>/resource/idl/QueuingServiceTypes.idl`.

The operational status provides multiple counters describing the status of the SharedReaderQueue.

Return Value:
Upon success, this command returns RTI_SERVICE_COMMAND_REPLY_OK in the `retcode` field of the reply. The operational status is sent in serialized form within the `octet_body` field in the CommandReply. If there is an error, this command returns RTI_SERVICE_COMMAND_REPLY_ERROR and the field `string_body` contains a human-readable string describing the error.

Status Description:
The type of the SharedReaderQueue’s status can be found in the file `<NDDSHOME>/resource/idl/QueuingServiceTypes.idl`.

To deserialize the status from the CommandReply `octet_body` use the following operations:

- C. SharedReaderQueueStatusTypeSupport_deserialize_data_from_cdr_buffer()
Remote Commands in Queuing Service

- C++: `SharedReaderQueueStatusTypeSupport::deserialize_data_from_cdr_buffer()
- C++/CLI: `SharedReaderQueueStatusTypeSupport::deserialize_data_from_cdr_buffer()
- C#: `SharedReaderQueueStatusTypeSupport::deserialize_data_from_cdr_buffer()
- Java: `SharedReaderQueueStatusTypeSupport::deserialize_data_from_cdr_buffer()

When generating code for `QueuingServiceTypes.idl` in C, C++, and .NET, make sure you use the `-unboundedSupport` command-line option.

### 5.4.5 Get Service Data

The following command is used to get the ServiceData that provides a sequence of SharedReaderQueueData. This command provides a way to query all the SharedReaderQueues hosted in a service instance.

```
GET <target_queuing_service> /data
```

**Return Value:**

Upon success, this command returns `RTI_SERVICE_COMMAND_REPLY_OK` in the `retcode` field of the reply. The ServiceData is sent in serialized form within the field `octet_body` in the CommandReply. If there is an error, this command returns `RTI_SERVICE_COMMAND_REPLY_ERROR` and the field `string_body` contains a human-readable string describing the error.

**Service Data:**

The type of the ServiceData can be found in the file `<NDDSHOME>/resource/idl/QueuingServiceTypes.idl`.

```c
struct SharedReaderQueueData {
    /* Fully qualified name of the SharedReaderQueue within the XML file */
    string<NAME_MAX_LENGTH> queue_name; // @key
    string<NAME_MAX_LENGTH> topic_name;
}; //@Extensibility MUTABLE_EXTENSIBILITY

struct ServiceData {
    sequence<SharedReaderQueueData> shared_reader_queue_data_list;
}; //@Extensibility MUTABLE_EXTENSIBILITY
```

To deserialize the ServiceData from the CommandReply `octet_body`, use the following operations:

- C: `ServiceDataTypeSupport::deserialize_data_from_cdr_buffer()`
- C++: `ServiceDataTypeSupport::deserialize_data_from_cdr_buffer()`
- C++/CLI: `ServiceDataTypeSupport::deserialize_data_from_cdr_buffer()`
- C#: `ServiceDataTypeSupport::deserialize_data_from_cdr_buffer()`
- Java: `ServiceDataTypeSupport::deserialize_data_from_cdr_buffer()`

When generating code for `QueuingServiceTypes.idl` in C, C++, and .NET, make sure you use the `-unboundedSupport` command-line option.
5.4.6 Get Samples From a SharedReaderQueue

The following command is used to get one or more samples from the SharedReaderQueue using a condition. This is a multi-reply command in which the number of responses is equal to the number of samples satisfying the condition.

GET <target_queuing_service> <shared_reader_queue_resource_identifier>/message <sample_selector>

Parameters:

The <sample_selector> (see Sample Selector (Section 5.2.2)) is a SQL expression that specifies the set of samples that must be retrieved. This expression must be provided in the field string_body of the CommandRequest.

Return Value:

Upon success, this command returns X number of replies where X is the number of samples in the SharedReaderQueue satisfying the <sample_selector> expression. In each one of these replies the retcode field is set to RTI_SERVICE_COMMAND_REPLY_OK and the octet_body is initialized with the serialized sample in CDR format.

If there are no samples satisfying the <sample_selector>, the service returns one reply where the retcode field is set to RTI_SERVICE_COMMAND_REPLY_OK and the octet_body is empty.

In multi-reply commands, you can detect the last reply for a given command by inspecting the field flag in DDS_SampleInfo. For intermediate replies, the flag DDS_INTERMEDIATE_REPLY_SEQUENCE_SAMPLE is set. In the last reply this flag is not set.

Samples:

Each one of the samples returned by this command in the field octet_body of the reply is encapsulated in a Message type, which has the following definition:

```c
struct Message {
    MessageStatusKind status;
    SampleIdentity_t original_virtual_sample_identity;
    /* CDR-serialized content of the SharedReaderQueue sample */
    SampleBuffer_t sample_buffer;
}; //EXTENSIBILITY
```

The type can be found in the file <NDDSHOME>/resource/idl/QueuingServiceTypes.idl.

To deserialize the Message from the CommandReply octet_body use the following operations:

- C: MessageTypeSupport_deserialize_data_from_cdr_buffer()
- C++: MessageTypeSupport::deserialize_data_from_cdr_buffer()
- C++/CLI: MessageTypeSupport::deserialize_data_from_cdr_buffer()
- C#: MessageTypeSupport::deserialize_data_from_cdr_buffer()
- Java: MessageTypeSupport::deserialize_data_from_cdr_buffer()

The sample_buffer field in Message contains the serialized SharedReaderQueue’s sample. To deserialize the sample use the following operations (where <Foo> is the type of the SharedReaderQueue’s samples):

- C: <Foo>Support_deserialize_data_from_cdr_buffer()
- C++: <Foo>Support::deserialize_data_from_cdr_buffer()
- C++/CLI: <Foo>Support::deserialize_data_from_cdr_buffer()
- C#: <Foo>Support::deserialize_data_from_cdr_buffer()
- Java: <Foo>Support::deserialize_data_from_cdr_buffer()
When generating code for QueuingServiceTypes.idl in C, C++, and .NET, make sure you use the -unboundedSupport command-line option.

5.4.7 Create SharedSubscriber

The following command is used to create a SharedSubscriber:

CREATE <target_queuing_service> <domain_participant_resource_identifier> <xml_url>

Parameters:

- `<domain_participant_resource_identifier>` is the resource identifier for the DomainParticipant that will contain the SharedSubscriber.
- `<xml_url>` contains an XML snippet containing the SharedSubscriber configuration. A full file (starting with `<dds>...`) is not valid. For example:

```
str://"<shared_subscriber name="SharedSubscriber_1"...>
... </shared_subscriber>"
```

Return Value:

Upon success, this command returns RTI_SERVICE_COMMAND_REPLY_OK in the retcode field of the reply. Otherwise, this command returns one of the following values, and the field string_body contains a human-readable string describing the error:

- RTI_SERVICE_COMMAND_REPLY_ALREADY_EXISTS, if the SharedSubscriber already exists with a different configuration.
- RTI_SERVICE_COMMAND_REPLY_DUPLICATED, if the SharedSubscriber already exists with the same configuration.
- RTI_SERVICE_COMMAND_REPLY_ERROR, for any other creation error.

5.4.8 Delete SharedSubscriber

The following command is used to delete a SharedSubscriber:

DELETE <target_queuing_service> <shared_subscriber_resource_identifier>

Return Value:

Upon success, this command returns RTI_SERVICE_COMMAND_REPLY_OK in the retcode field of the reply. Otherwise, this command returns RTI_SERVICE_COMMAND_REPLY_ERROR and the field string_body contains a human-readable string describing the error.

5.4.9 Shutdown

The following command is used to shutdown a Queuing Service process:

SHUTDOWN <target_queuing_service>

Return Value:

Upon success, this command returns RTI_SERVICE_COMMAND_REPLY_OK in the retcode field of the reply, and the shutdown sequence is initiated in the remote service process. Otherwise, this command returns one of the following values, and the field string_body contains a human-readable string describing the error:

- RTI_SERVICE_COMMAND_REPLY_UNSUPPORTED, if the service is deployed as a library or deamon.
- RTI_SERVICE_COMMAND_REPLY_ERROR, for any other error.
5.5 Accessing Queuing Service from a Connext DDS application

You can create a DataWriter for the command topic to write Queuing Service administration commands and create a DataReader for the response topic to receive responses.

A more powerful and easier way is to use the Request-Reply API (only available with Connext DDS Professional). You can create a Requester for these topics that will write command requests and wait for replies.

The QoS configurations of your DataWriter and DataReader, or your Requester (if you are using the Request-Reply API), must be compatible with the one used by Queuing Service (see how this is configured in Configuring Administration (Section 3.3.3)).

The following C# example uses the Connext DDS Professional Request-Reply API to send commands and receive replies. This example shows a C# application that creates a Requester that can communicate with the Queuing Service remote administration server. It sends a request (a Queuing Service remote command) to create a SharedReaderQueue. The request receives a reply with the result for that command.

```csharp
using System;
using System.Collections.Generic;
using System.Text;
using RTI.Service.Admin;
using DDS;
using RTI.Connext.RequestReply;
using RTI.Connext.Infrastructure;

/**
 * How to use the Queuing Service administration through a Requester
 */
public class CommandExample
{
    private static readonly Duration_t MAX_WAIT = Duration_t.from_seconds(10);

    public static void Main(string[] args)
    {
        // Create DomainParticipant
        DomainParticipant participant =
            DomainParticipantFactory.get_instance()
            .create_participant(1,
            DomainParticipantFactory.PARTICIPANT_QOS_DEFAULT, null,
            StatusMask.STATUS_MASK_NONE);

        if (participant == null)
        {
            throw new SystemException("Participant creation failed");
        }

        try
        {
            // Accessing Queuing Service from a Connext application
            // Create requester for Routing Service remote-admin topics //
            Requester<CommandRequest, CommandReply> requester =
```
new Requester<CommandRequest, CommandReply>(
    new RequesterParams(participant,
        CommandRequestTypeSupport.get_instance(),
        CommandReplyTypeSupport.get_instance()).
    SetRequestTopicName(
        COMMAND_REQUEST_TOPIC_NAME.VALUE).
    SetReplyTopicName(COMMAND_REPLY_TOPIC_NAME.VALUE));

try
{
    System.Console.WriteLine("Waiting to discover Queuing Service");
    InstanceHandleSeq handles = new InstanceHandleSeq();
    while (handles.length == 0)
    {
        requester.RequestDataWriter.
            get_matched_subscriptions(handles);
        System.Threading.Thread.Sleep(1000);
    }
    System.Console.WriteLine("Matched subscription");
    //
    // Send create command
    //
    WriteSample<CommandRequest> request =
        requester.CreateRequestSample();

    request.Data.service_name = "QueuingService_1";
    request.Data.action = CommandActionKind.RTI_SERVICE_COMMAND_ACTION_CREATE;
    request.Data.resource_identifier =
        "/domain_participant/DomainParticipant/shared_subscriber/SharedSubscriber";

    request.Data.string_body =
        "str:\"<shared_reader_queue name="SharedReaderQueue_1" session="session">
        \n        <topic_name>SharedReaderQueueTopic</topic_name>
        <type_name> SharedReaderQueueType</type_name>
        \n        </shared_reader_queue>\"; 

    requester.SendRequest(request);

    // Receive the reply 
    Sample<CommandReply> reply = requester.CreateReplySample();
    if (!requester.ReceiveReply(reply, MAX_WAIT))
    {
        throw new SystemException("Reply not received");
    }
    System.Console.WriteLine("Received reply: " + reply.Data.string_body);
}
finally
{
    requester.Dispose();
}
}
finally
{
    participant.delete_contained_entities();
}
DomainParticipantFactory.get_instance().
delete_participant(ref participant);
You can monitor Queuing Service remotely by subscribing to special topics. By subscribing to these topics, any Connext DDS application can receive information about the configuration and operational status of Queuing Service.

Being able to monitor the state of a Queuing Service instance is an important tool that allows you to detect problems. For example, looking at the enqueue throughput of a SharedReaderQueue you might see that the queue is receiving a lot of traffic and you may want to put that queue in its own session.

There are two kinds of monitoring data for an entity (for example, a SharedReaderQueue):

- **Entity data** provides information about the configuration of the entity. For example, the service data contains a list of the SharedReaderQueues contained in the service. Entity data information is updated every time there is a configuration change that affects that data.

- **Entity status** provides information about the operational status of an entity. This kind of information changes continuously and is computed and published periodically. For example, the SharedReaderQueue status contains information such as the SharedReader-Queue’s latency and throughput.

Queuing Service only publishes entity status for SharedReaderQueues. Entity data can be accessed using remote administration commands (See Chapter 5: Administering Queuing Service from a Remote Location.)

### 6.1 Enabling Publish-Subscribe Monitoring Data

By default, remote publish-subscribe monitoring is disabled in Queuing Service for security reasons. To enable remote monitoring, you can use the `<monitoring>` tag (see Configuring Publish-Subscribe Monitoring (Section 3.3.4.2)).

When remote publish-subscribe monitoring is enabled, Queuing Service creates:

- 1 DomainParticipant
- 1 Publisher
- 1 DataWriter to publish status data for SharedReaderQueues

The QoS values for these entities are described in Configuring Publish-Subscribe Monitoring (Section 3.3.4.2).
6.2 Status Information for a SharedReaderQueue

The topic that publishes SharedReaderQueue status is called rti/queuing_service/monitoring/shared_reader_queue.

The registered type name for the topic is RTI::QueuingService::Monitoring::SharedReaderQueueStatus.

The type definition of the SharedReaderQueue status is called SharedReaderQueueStatus and it can be found in the file <NDDSHOME>/resource/idl/QueuingServiceTypes.idl.

Queuing Service reports multiple statistics as part of the SharedReaderQueue status. Some of these statistics are counters such as the number of samples received by a SharedReaderQueue and other statistics are statistics variables such as the number of samples enqueued per second in a SharedReaderQueue.

To see how statistics variable are calculated, see Configuring Statistics Calculation Process (Section 3.3.4.3).
Chapter 7  High Availability

For high availability, Queuing Service can be configured to replicate both, the content of the SharedReaderQueues and the service configuration.

7.1  SharedReaderQueue Replication

By default, SharedReaderQueues within a Queuing Service instance are not replicated. SharedReaderQueues can optionally be replicated across multiple instances of Queuing Service running in the same or different nodes. See Figure 7.1

Figure 7.1  Replicating SharedReaderQueues

7.1.1  SharedReaderQueue Replication Protocol

Each replicated SharedReaderQueue consists of one master and multiple slaves. Only the master SharedReaderQueue distributes messages to the QueueConsumers DataReaders. When the master goes away the most up-to-date slave is promoted into master.
The replication protocol has four different phases:

1. Sample replication
2. Enqueue
3. Consumer assignment
4. Delivery

7.1.1.1 Sample Replication Phase

During this phase, the samples published by a QueueProducer's DataWriter are distributed to all replicas (master and slaves). There are two ways to do this:

1. The QueueProducer’s DataWriter sends directly the samples to all the replicas. This is the preferred way to distribute the sample as it provides the best performance, especially with the usage of multicast. See Figure 7.2.

Figure 7.2 Direct Sample Distribution
2. The QueueProducer's DataWriter sends the samples to only a subset of the replicas, usually one. Then the replicas that receive the samples broadcast these samples to all the other replicas, as seen in Figure 7.3.

Figure 7.3 Relayed Sample Distribution

In this release, the decision of whether or not a replica should broadcast the received samples to the other replicas is taken by the QueueProducer’s application on a per-sample basis by marking the sample with the flag DDS_REPLICATE_SAMPLE. This can be done by using the DataWriter’s write_w_params() operation and setting the bit DDS_REPLICATE_SAMPLE in the flag field of WriteParams_t.

7.1.1.2 Enqueue Phase

During the enqueue phase, the master makes sure at least a quorum of the most up-to-date replicas (including itself) have received a sample before moving the sample to the ENQUEUE state (see Sample Lifecycle In Queuing Service (Section 2.9)).

The number of replicas in the quorum is defined as the lowest integer that is higher than half of the expected number of replicas. The expected number of replicas must be known in advance and it is configured using the XML tag <queue_instances> under <replication_settings> (see SharedReaderQueue Replication Configuration (Section 7.1.3)).

After the sample is moved to the ENQUEUE state, the master and slaves send an AppAck message to the QueueProducer indicating that the sample has been successfully enqueued. The response data of the AppAck message for successfully enqueued samples will be a single byte set to 1. Positive AppAck messages are global AppAck messages. Therefore, when monitoring AppAck messages, the QueueProducer can assume that a sample has been successfully enqueued as soon as it receives a positive acknowledgment from any of the replicas (master or slaves).
If there is no a quorum of up-to-date replicas that are able to enqueue the sample, the replicas will send an AppAck message to the QueueProducer's DataWriter, where the response is set to 0. Negative AppAck messages are local messages. In order to consider a message as not enqueued, a QueueProducer must receive a negative AppAck from all replicas.

To make this decision easier, you can use the DataWriter's is_sample_app_acknowledged() operation—it returns TRUE when a sample has been application acknowledged (negatively or positively) by all replicas that were alive when the sample was published. If the QueueProducer has not received a positive AppAck message for a sample and the is_sample_app_acknowledged() returns TRUE, the sample can be considered not enqueued. At this point it is responsibility of the application to decide whether or not to republish the sample.

7.1.1.3 Consumer Assignment Phase

During the consumer assignment phase the master selects a QueueConsumer as the destination for a message according to the distribution policy configured for the SharedReaderQueue (see Selecting a QueueConsumer for a Sample (Section 2.10)).

After the QueueConsumer has been selected, the master notifies all the slaves about this selection. Then, there are two possibilities:

1. The master sends the sample to the QueueConsumer immediately.
2. The master waits until it gets confirmation from the quorum of most up-to-date slaves indicating that they received the assignment before it sends the sample to the QueueConsumer.

This behavior can be configured using the XML tag <synchronize_consumer_assignment> under <replication_settings> (see SharedReaderQueue Replication Configuration (Section 7.1.3)).

If the master goes away, the slave promoted to master will try first to send the samples to the assigned QueueConsumer if this QueueConsumer is still in the system. These samples will be marked with the DDS_REDELIVERED_SAMPLE flag.

7.1.1.4 Delivery Phase

After a QueueConsumer sends an application-level acknowledgment to the master indicating that a sample has been processed successfully, the master notifies all the slaves about this decision and it removes the sample from the SharedReaderQueue. When the slaves receive this notification they also remove the sample from their SharedReaderQueues.

7.1.2 SharedReaderQueue Master Election Protocol

When the master for a SharedReaderQueue goes away the most up-to-date slave is promoted into master.

How fast the loss of the master is detected depends on a master timeout period configurable using the XML tag <master_timeout> under <replication_settings> (See SharedReaderQueue Replication Configuration (Section 7.1.3)).

If a slave does not receive messages from the master during a period greater than the master timeout, it initiates a voting mechanism to select a new master.

While the new master election is in progress, the samples sent by QueueProducers will be rejected. The QueueProducer will receive AppAck messages from all replicas with the response set to 0.
7.1.3 SharedReaderQueue Replication Configuration

You can choose between replicating all the SharedReaderQueues within a service or replicating individual SharedReaderQueues.

To replicate all the SharedReaderQueues within a service, you can set the `<shared_reader_queue_replication>` tag within `<queuing_service>/<service_qos>`. Replication is automatically enabled when you use this tag. It also allows you to configure the replication protocol.

Table 7.1, SharedReaderQueue Replication Tags describes the tags allowed within a `<shared_reader_queue_replication>` tag.

You can also replicate individual SharedReaderQueues by using the `<replication>` tag under `<shared_reader_queue>/<queue_qos>` (see Table 7.2, Replication Tags).

Table 7.1 SharedReaderQueue Replication Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;shared_reader_queue_replication&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;enabled&gt;</code></td>
<td>Enables/disables replication for all SharedReaderQueues in the service. You can override this behavior on a per SharedReaderQueue basis by setting <code>&lt;replication&gt;</code> under <code>&lt;shared_reader_queue&gt;/&lt;queue_qos&gt;</code>. Default: true</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;replication_settings&gt;</code></td>
<td>Configures the replication protocol. See Table 7.3, Replication Settings Tags. Default: If not set, replication settings are inherited from the settings in <code>&lt;replication_settings&gt;</code> under <code>&lt;queuing_service&gt;</code>.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

Table 7.2 Replication Tags

<table>
<thead>
<tr>
<th>Tags within <code>&lt;replication&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;enabled&gt;</code></td>
<td>Enables/disables replication for the SharedReaderQueue Default: true</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;replication_settings&gt;</code></td>
<td>Configures the replication protocol. See Table 7.3, Replication Settings Tags. Default: If not set, replication settings are inherited as follows: First, from the settings in <code>&lt;replication_settings&gt;</code> under <code>&lt;queuing_service&gt;/&lt;service_qos&gt;</code>; <code>&lt;shared_reader_queue_replication&gt;</code> Second, from the settings in <code>&lt;replication_settings&gt;</code> under <code>&lt;queuing_service&gt;</code></td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

The replication protocol is configured using the `<replication_settings>` tag; see Table 7.3, Replication Settings Tags.
7.1.3.1 Protocol Information Exchange

The replication of SharedReaderQueues requires the exchange of status information among replicas. This is done by creating a DataWriter and a DataReader per SharedReaderQueue to publish and subscribe to this information.

The QoS for these entities can be configured using the tags `<update_datawriter_qos>` and `<update_datareader_qos>` under the `<shared_reader_queue>` tag; see Table 3.14, SharedReaderQueue Tags.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;replication_settings&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;queue_instances&gt;</code></td>
<td>The number of expected replicas (including the master) for a SharedReaderQueue</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default: 2</td>
<td></td>
</tr>
<tr>
<td><code>&lt;master_timeout&gt;</code></td>
<td>A new master election process will be initiated if the master does not send messages to the replicas before this timeout expires. Example:</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td><code>&lt;master_timeout&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;sec&gt;5&lt;/sec&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>&lt;nanosec&gt;0&lt;/nanosec&gt;</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Default: 5 seconds</td>
<td></td>
</tr>
<tr>
<td><code>&lt;synchronize_consumer_assignment&gt;</code></td>
<td>Indicates if the master must wait for the slaves to receive the QueueConsumer assignment before sending a sample to the selected QueueConsumer.</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default: false</td>
<td></td>
</tr>
<tr>
<td><code>&lt;sample_timeout&gt;</code></td>
<td>Configures the maximum amount of time that a sample can be in a replica’s SharedReaderQueue without having reached quorum. After this time, the sample is removed from the SharedReaderQueue, the replica sends an AppAck message to the QueueProducer (with the response set to 0), and the replica notifies the master about this event. Notice that the sample is not sent to the DeadLetterSharedReaderQueue. This timeout is needed to avoid situations in which a sample stays in the replicas’ SharedReaderQueues permanently. This could happen if for some reason one of the replicas participating in the quorum did not receive a sample from the QueueProducer. Under this circumstance, the sample would not be able to be enqueued with quorum and it would stay in the SharedReaderQueues of the replicas that received the sample indefinitely.</td>
<td>0 or 1</td>
</tr>
<tr>
<td></td>
<td>Default: 7 seconds, measured from the enqueue time</td>
<td></td>
</tr>
</tbody>
</table>
7.2 Configuration Replication

By default, the service configuration is not replicated. Enabling configuration replication between a set of Queuing Service instances (replication cluster) will require:

- Enabling remote administration and using the same remote administration domain ID for each one of the Queuing Service instances participating in the configuration replication process. The administration domain ID can be configured using the command-line option `-remoteAdministrationDomainId` (see Table 4.1, RTI Queuing Service Command-Line Options) or the XML tag `<administration>/domain_id>`.

- Assigning an application name to each one of the Queuing Service instances using the command-line option `-appName`.

- Setting the tag `<configuration_replication>` under `<service_qos>` in the configuration file (see Chapter 7: Administering Queuing Service from a Remote Location). This will create a special SharedReaderQueue for configuration replication that runs in its own DomainParticipant.

- [Optional] Using the `-cfgRemote` command-line option in combination with `-remoteAdministrationDomainId` to obtain the initial configuration from other running instances. Set `-remoteAdministrationDomainId` to the administration domain ID that will be used to send remote commands. If you do not use `-cfgRemote`, the service will not get the initial configuration remotely and it will start from the provided file.

When replication is enabled, remote administration commands that change the service configuration, such as adding or removing a SharedReaderQueue, should be sent to all the Queuing Service instances by using `<CommonPrefix>` as the target queuing service (field `service_name` in CommandRequest) (see Chapter 5: Administering Queuing Service from a Remote Location). In the above example, the field `service_name` would be set to `Cluster_1*`. The application sending the command will receive a response from each one of the members in the cluster, confirming the successful execution of the command.

Notice that an application could still send a command to multiple Queuing Service instances without enabling replication. The difference in this case is that the final configuration may not be consistent across instances if multiple applications send remote commands at the same time. By enabling replication using the tag `<configuration_replication>`, we guarantee configuration consistency across all the instances in the cluster.

7.2.1 SharedReaderQueue for Configuration Replication

To enable configuration replication, you must use the `<configuration_replication>` tag under `<service_qos>`. When this occurs, Queuing Service creates a special SharedReaderQueue that is used to replicate the remote administration commands across all the instances in the replication.
cluster. This SharedReaderQueue is replicated and the replication settings are configured using the `<replication_settings>` flag under `<configuration_replication>`. Table 7.4, Configuration Replication Tags describes the tags allowed within a `<configuration_replication>` tag.

<table>
<thead>
<tr>
<th>Tags within <code>&lt;configuration_replication&gt;</code></th>
<th>Description</th>
<th>Number of Tags Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;enabled&gt;</code></td>
<td>Enables/disables configuration replication. Default: true</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;participant_qos&gt;</code></td>
<td>Configures the DomainParticipant QoS for configuration replication. This DomainParticipant runs on the administration domain ID. If the tag is not defined, Queuing Service will use the Connext DDS defaults.</td>
<td>0 or 1</td>
</tr>
<tr>
<td><code>&lt;replication_settings&gt;</code></td>
<td>Configures the configuration replication protocol. See Table 7.3, Replication Settings Tags Default: If not set, the replication settings are inherited from the settings in <code>&lt;replication_settings&gt;</code> under <code>&lt;queuing_service&gt;</code>.</td>
<td>0 or 1</td>
</tr>
</tbody>
</table>

### 7.3 Replication Clusters

A replication cluster is a set of Queuing Service instances that coordinate with each other to replicate SharedReaderQueues and/or the service configuration. Instances in different clusters are isolated from each other.

For SharedReaderQueue replication, all instances within a cluster must have a `<domain_participant>` with the same `<domain_id>` (see Configuring DomainParticipants (Section 3.3.6)).

For service configuration replication, all instances within a cluster must use the same `<domain_id>` for remote administration (See Configuring Administration (Section 3.3.3))
Figure 7.4  Replication Cluster
Chapter 8 Queuing Service Wrapper API

RTI Connext DDS provides a wrapper API to make it easier to interact with Queuing Service.

In this release, the wrapper API is only supported in the .NET API and is located in the namespace RTI.Connext.Queuing.

Important: The wrapper API is only available with the Connext DDS Professional, Evaluation, and Basic package types (it is not available with the Core package type).

8.1 QueueProducer Wrapper

To simplify the use and configuration of a DataWriter to send samples to a SharedReaderQueue, Connext DDS provides an abstraction, QueueProducer<MessageType>, which wraps the DataWriter and provides additional services such as an operation to detect if there is a matching SharedReaderQueue or an operation to wait for application-level acknowledgement after sending a sample.

The Connext DDS API Reference HTML documentation contains the full API documentation for the QueueProducer. Under the Modules tab, navigate to RTI Connext DDS API Reference, RTI Connext Messaging API Reference, Queuing Pattern, QueueProducer.

8.2 QueueConsumer Wrapper

To simplify the use and configuration of a DataReader to receive samples from a SharedReaderQueue, Connext provides an abstraction, QueueConsumer<MessageType>, which wraps the DataReader and provide additional services such as an operation to detect if there is a matching SharedReaderQueue or a blocking operation to receive samples.

The Connext API Reference HTML documentation contains the full API documentation for the QueueProducer. Under the Modules tab, navigate to RTI Connext DDS API Reference, RTI Connext Messaging API Reference, Queuing Pattern, QueueConsumer.

8.3 QueueRequester Wrapper

To simplify the use and configuration of the DataReader and DataWriter in the requester application, Connext provides an abstraction, QueueRequester<MessageRequestType, MessageRep-
QueueReplier Wrapper

Type>, which wraps the DataReader and DataWriter usage and provide additional services such as an operation to wait for the response for a given request.

The Connext API Reference HTML documentation contains the full API documentation for the QueueProducer. Under the Modules tab, navigate to RTI Connext DDS API Reference, RTI Connext Messaging API Reference, Queuing Pattern, QueueRequester.

8.4 QueueReplier Wrapper

To simplify the use and configuration of the DataReader and DataWriter in the replier application, Connext provides an abstraction, QueueReplier<MessageRequestType, MessageReplyType>, which wraps the DataReader and DataWriter usage.

The Connext API Reference HTML documentation contains the full API documentation for the QueueProducer. Under the Modules tab, navigate to RTI Connext DDS API Reference, RTI Connext Messaging API Reference, Queuing Pattern, QueueReplier.
Chapter 9  Communication Using TCP Transport

Queuing Service, and the applications that interact with it, can be configured to communicate with each other using the TCP transport distributed with Connext DDS. The transport can be configured via XML using the PropertyQosPolicy of the Queuing Service's DomainParticipants and the applications' DomainParticipants.

This chapter explains how to use and configure TCP communications with Queuing Service. This chapter does not intend to provide an exhaustive explanation of the TCP transport and all of its configuration properties. For details on the TCP transport, see the RTI Connext DDS Core Libraries and Utilities User's Manual.

The TCP transport distributed with Connext DDS can be used to address multiple communication scenarios that range from simple communication within a single LAN, to complex communication scenarios across LANs where NATs and firewalls may be involved.

The next sections explain how to configure and use the TCP transport to communicate with Queuing Service in some typical scenarios.

9.1  Asymmetric TCP Communication With Queuing Service

In this scenario, Queuing Service is behind a NAT/Firewall and the QueueProducers, QueueConsumers, and Remote Administration applications run outside the NAT. TCP connections can be initiated only by applications running outside the NAT.

Figure 9.1 shows how to configure the system to communicate using the TCP transport. Notice that it is not necessary to set NDDS_DISCOVERY_PEERS in the Queuing Service instance because the connections are initiated from the applications running outside the NAT. In this example, Queuing Service instantiates two instances of the TCP transport: one for administration and one for SharedReaderQueue traffic. Each instance uses a separate TCP port.
The following XML snippet shows how to configure the TCP transport in *Queuing Service*. For convenience, the participant QoS in the administration and SharedReaderQueue domains inherits from a common QoS profile `TCPLibrary::TCPProfile`.

```xml
<qos_library name="TCPLibrary">
    <qos_profile name="TCPProfile">
        <participant_qos>
            <property>
                <value>
                    <element>
                        <name>dds.transport.load_plugins</name>
                        <value>dds.transport.tcp</value>
                    </element>
                    <element>
                        <name>dds.transport.tcp.library</name>
                        <value>nddstransporttcp</value>
                    </element>
                    <element>
                        <name>dds.transport.tcp.parent.classid</name>
                        <value>NDDS_TRANSPORT_CLASSID_TCPV4_WAN</value>
                    </element>
                    <element>
                        <name>dds.transport.tcp.create_function</name>
                        <value>NDDS_Transport_TCPv4_create</value>
                    </element>
                </value>
            </property>
        </participant_qos>
    </qos_profile>
</qos_library>

<queuing_service name="Service">
    <administration>
        <domain_id>1</domain_id>
        <participant_qos base_name="TCPLibrary::TCPProfile">
            <property>
                <value>
                    <element>
                        ...
                    </element>
                </value>
            </property>
        </participant_qos>
    </administration>
</queuing_service>
```
The following XML snippet shows how to configure the applications running outside the NAT.

```
<participant_qos>
  <property>
    <element>
      <name>dds.transport.load_plugins</name>
      <value>dds.transport.tcp</value>
    </element>
    <element>
      <name>dds.transport.tcp.library</name>
      <value>nddstransporttcp</value>
    </element>
    <element>
      <name>dds.transport.tcp.parent.classid</name>
      <value>NDDS_TRANSPORT_CLASSID_TCPV4_WAN</value>
    </element>
    <element>
      <name>dds.transport.tcp.create_function</name>
      <value>NDDS_Transport_TCPv4_create</value>
    </element>
    <element>
      <name>dds.transport.tcp.server_bind_port</name>
      <value>0</value>
    </element>
  </property>
</participant_qos>
```
9.2 Asymmetric TCP Communication with Queuing Service And Replication

In this scenario, one of more instances of Queuing Service are behind a NAT/Firewall and the QueueProducers, QueueConsumers, and Remote Administration applications run outside the NAT. The Queuing Service instances are configured to replicate SharedReaderQueues and configuration.

Figure 9.2 shows how to configure the system to communicate using the TCP transport. This includes communication with the applications running outside the NAT and communication between the Queuing Service instances.

![Asymmetric TCP Configuration With Replication](image)

In a basic scenario that does not include configuration replication, a Queuing Service instance creates two DomainParticipants:

1. The first DomainParticipant is used to communicate with QueueProducers and QueueConsumers. This DomainParticipant is also used to exchange SharedReaderQueue synchronization information between Queuing Service instances. To configure QoS of this DomainParticipant, use the `<domain_participant>/<participant_qos>` tag (see Configuring DomainParticipants (Section 3.3.6)).

2. The second DomainParticipant is used to receive remote administration commands. To configure its QoS, use the `<administration>/<participant_qos>` tag (see Configuring Administration (Section 3.3.3)).
When Queuing Service is configured to replicate configuration, it creates one more DomainParticipants to replicate the configuration. The QoS of this DomainParticipants is configured using \texttt{<configuration_replication>/participant_qos} (see SharedReaderQueue for Configuration Replication (Section 7.2.1)). All this DomainParticipants must be configured to use TCP.

This TCP communication scenario will require creating two instances of the TCP transport in each one of the DomainParticipants created by the Queuing Service (QS) instances:

- The first instance runs in asymmetric mode and is used to allow the Queuing Services to communicate with Producers, Consumers, and Remote Administration applications.
- The second instance runs in symmetric mode and is used for communication between Queuing Services. Symmetric mode means that each service will create a server socket that other services will use to establish connections.

The following XML snippet shows how to configure the TCP transport in Queuing Service

```xml
<qos_library name="TCPLibrary">
  <qos_profile name="TCPProfile">
    <participant_qos>
      <property>
        <element>
          <name>dds.transport.load_plugins</name>
          <value>dds.transport.tcp,dds.transport.tcp2</value>
        </element>
        <element>
          <name>dds.transport.tcp.library</name>
          <value>nddstransporttcp</value>
        </element>
        <element>
          <name>dds.transport.tcp.parent.classid</name>
          <value>NDDS_TRANSPORT_CLASSID_TCPV4_WAN</value>
        </element>
        <element>
          <name>dds.transport.tcp.create_function</name>
          <value>NDDS_Transport_TCPv4_create</value>
        </element>
      </property>
    </participant_qos>
  </qos_profile>
</qos_library>

<queuing_service name="Service">
  <administration>
    <domain_id>1</domain_id>
    <participant_qos base_name="TCPLibrary::TCPProfile">
      ...
    </participant_qos>
  </administration>
</queuing_service>
```
Asymmetric TCP Communication with Queuing Service And Replication

<property>
  <element>
    <name>dds.transport.tcp.server_bind_port</name>
    <value>15002</value>
  </element>
  <element>
    <name>dds.transport.tcp.public_address</name>
    <value>18.181.0.32:15002</value>
  </element>
  <element>
    <name>dds.transport.tcp2.server_bind_port</name>
    <value>15003</value>
  </element>
  <element>
    <name>dds.transport.tcp2.public_address</name>
    <value>192.168.5.11:15003</value>
  </element>
</property>

<configuration_replication>
  <participant_qos base_name="TCPLibrary::TCPProfile">
    <property>
      <value>
        <element>
          <name>dds.transport.tcp.server_bind_port</name>
          <value>15004</value>
        </element>
        <element>
          <name>dds.transport.tcp.public_address</name>
          <value>18.181.0.32:15004</value>
        </element>
        <element>
          <name>dds.transport.tcp2.server_bind_port</name>
          <value>15005</value>
        </element>
        <element>
          <name>dds.transport.tcp2.public_address</name>
          <value>192.168.5.11:15005</value>
        </element>
      </value>
    </property>
  </participant_qos>
</configuration_replication>

<domain_participant name="DomainParticipant">
  <domain_id>0</domain_id>
  <participant_qos base_name="TCPLibrary::TCPProfile">
    <property>
      <value>
        <element>
          <name>dds.transport.tcp.server_bind_port</name>
          <value>15000</value>
        </element>
      </value>
    </property>
  </participant_qos>
</domain_participant>
The following XML snippet shows how to configure the applications running outside the NAT.

```xml
<participant_qos>
  <property>
    <value>
      <element>
        <name>dds.transport.load_plugins</name>
        <value>dds.transport.tcp</value>
      </element>
      <element>
        <name>dds.transport.tcp.library</name>
        <value>nddstransporttcp</value>
      </element>
      <element>
        <name>dds.transport.tcp.parent.classid</name>
        <value>NDDS_TRANSPORT_CLASSID_TCPV4_WAN</value>
      </element>
      <element>
        <name>dds.transport.tcp.create_function</name>
        <value>NDDS_Transport_TCPv4_create</value>
      </element>
      <element>
        <name>dds.transport.tcp.server_bind_port</name>
        <value>0</value>
      </element>
    </value>
  </property>
</participant_qos>
```