RTI Connext DDS QoS Reference Guide

This document provides a high-level view of all the RTI® Connext® DDS QoS Policies in release 6.1.0.

QoS Policy List

See Legend below table.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AsynchronousPublisher</td>
<td>Configures mechanism that sends user data in a separate middleware thread</td>
<td>(Y)</td>
<td>BC</td>
<td></td>
<td>P</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Availability</td>
<td>Configures Collaborative DataWriters and Durable Subscriptions</td>
<td>A</td>
<td></td>
<td>R</td>
<td>W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Batch</td>
<td>Configures mechanism for collecting multiple data samples to be sent in a single network packet</td>
<td>(Y)</td>
<td>BE *</td>
<td></td>
<td>W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Database</td>
<td>Configures threads and resource limits that control internal Connext DDS data storage</td>
<td>BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DataReaderProtocol, DataWriter-Protocol</td>
<td>Configures DDS on-the-wire protocol (RTPS), along with WireProtocol QoS for Reliable Topics</td>
<td>(Y)</td>
<td>BC</td>
<td></td>
<td>R,W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DataReaderResourceLimits, DataWriterResourceLimits</td>
<td>Configures how DataReaders/DataWriters allocate and use physical memory for Connext DDS specific internal resources</td>
<td>(Y)</td>
<td>BC</td>
<td></td>
<td>R,W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>DataRepresentation</td>
<td>Specifies which CDR versions and data compression algorithms are offered and requested</td>
<td>Y</td>
<td>BE *</td>
<td></td>
<td>T,R,W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>DataTag</td>
<td>Associates name/value string pairs with DataWriters/DataReaders that may be used by Access Control plugin</td>
<td>Y</td>
<td>BC</td>
<td></td>
<td></td>
<td>R,W</td>
<td></td>
</tr>
<tr>
<td>Deadline</td>
<td>Maximum duration within which an instance is expected to be updated</td>
<td>Y</td>
<td>A*</td>
<td>Y</td>
<td>Y</td>
<td>T,R,W</td>
<td></td>
</tr>
<tr>
<td>DestinationOrder</td>
<td>Controls how RTI deals with data from multiple DataWriters for the same instance; &quot;by reception timestamp&quot; or &quot;by source timestamp&quot;</td>
<td>(Y)</td>
<td>BE*</td>
<td>Y</td>
<td>Y</td>
<td>T,R,W</td>
<td></td>
</tr>
<tr>
<td>Discovery</td>
<td>Configures which DomainParticipants to contact, how to automatically discover and connect with them</td>
<td>(Y)</td>
<td>BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
</tr>
<tr>
<td>DiscoveryConfig</td>
<td>Configures how DomainParticipants allocate and use physical memory for internal resources</td>
<td>(Y)</td>
<td>BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
</tr>
<tr>
<td>DomainParticipantResourceLimits</td>
<td>Configures how DomainParticipants allocate and use physical memory for internal resources</td>
<td>(Y)</td>
<td>BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
</tr>
<tr>
<td>Durability</td>
<td>Specifies if Connext DDS will store and deliver previously published data to new/late-joining Readers</td>
<td>(Y)</td>
<td>BE*</td>
<td>Y</td>
<td></td>
<td>T,R,W</td>
<td></td>
</tr>
<tr>
<td>DurabilityService</td>
<td>Configures external Persistence Service for Writers with PERSISTENT or TRANSIENT Durability</td>
<td></td>
<td>BE</td>
<td></td>
<td></td>
<td>T,W</td>
<td></td>
</tr>
<tr>
<td>EntityFactory</td>
<td>Controls Entity behavior as a factory for other entities (i.e., if child entities are created enabled)</td>
<td>Y</td>
<td>A</td>
<td></td>
<td></td>
<td>F,D,P,S</td>
<td></td>
</tr>
<tr>
<td>EntityName</td>
<td>Assigns a name to an Entity, pairs DomainParticipant with a Durable Subscription</td>
<td>(Y)</td>
<td>BE*</td>
<td></td>
<td></td>
<td>D,P,S,R,W</td>
<td>Y</td>
</tr>
<tr>
<td>Event</td>
<td>Configures the internal thread in a DomainParticipant that handles timed events</td>
<td></td>
<td>BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
</tr>
<tr>
<td>ExclusiveArea</td>
<td>Configures multi-thread concurrency and deadlock prevention capabilities</td>
<td></td>
<td>BC</td>
<td></td>
<td></td>
<td>P,S</td>
<td>Y</td>
</tr>
<tr>
<td>History</td>
<td>Controls how much data to store and how stored data is managed for the DataWriter-/DataReader within specified resource limits (see ResourceLimits)</td>
<td>(Y)</td>
<td>BE*</td>
<td>Y</td>
<td></td>
<td>T,R,W</td>
<td></td>
</tr>
<tr>
<td>LatencyBudget</td>
<td>Suggests to the middleware how much time is allowed to deliver data</td>
<td>A</td>
<td>Y</td>
<td></td>
<td></td>
<td>T,R,W</td>
<td></td>
</tr>
<tr>
<td>Lifespan</td>
<td>Specifies how long Connext DDS considers data sent by a user application to be valid</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>T,W</td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>Liveliness</td>
<td>Controls mechanism that allows DataReaders to detect when matching DataWriters become disconnected/dead</td>
<td>Y</td>
<td>BE *</td>
<td>Y</td>
<td>T,R,W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logging</td>
<td>Configures the Logging feature</td>
<td>A</td>
<td>F</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MultiChannel</td>
<td>Configures DataWriter to send data using different multicast groups (addresses) based on data value</td>
<td>BC</td>
<td>W</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ownership</td>
<td>Specifies if DataReader can receive new samples for instance of data from multiple DataWriters at same time</td>
<td>Y</td>
<td>BE *</td>
<td>Y</td>
<td>Y</td>
<td>T,R,W</td>
<td></td>
</tr>
<tr>
<td>OwnershipStrength</td>
<td>Specifies strength used to arbitrate among multiple DataWriters of same instance</td>
<td>Y</td>
<td>A *</td>
<td>Y</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Partition</td>
<td>Adds string IDs for finer-grained matching between DataWriters and DataReaders of same Topic</td>
<td>A</td>
<td>P,S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Presentation</td>
<td>Controls how Connext DDS presents data received by an application to DataReaders</td>
<td>BE</td>
<td>Y</td>
<td>Y</td>
<td>P,S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Profile</td>
<td>Configures how the DomainParticipantFactory loads XML documents containing QoS profiles</td>
<td>BC</td>
<td>F</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Property</td>
<td>Specifies name/value pairs used to configure parameters not exposed via formal QoS policies</td>
<td>(Y)</td>
<td>A</td>
<td></td>
<td>D,R,W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>PublishMode</td>
<td>Specifies whether data samples will be sent in the application thread or a middleware thread</td>
<td>Y</td>
<td>BC</td>
<td></td>
<td>W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>ReaderDataLifecycle</td>
<td>Controls how a DataReader manages the lifecycle of the data that it has received</td>
<td>A</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReceiverPool</td>
<td>Configures threads used to receive data from network transports (e.g., UDP) and deliver data to the application</td>
<td>BC</td>
<td>D</td>
<td></td>
<td>Y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reliability</td>
<td>Indicates whether samples lost by the network should be repaired by the middleware</td>
<td>(Y)</td>
<td>BE *</td>
<td>Y</td>
<td>T,R,W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>Limits amount of data cached by the middleware</td>
<td>(Y)</td>
<td>BE *</td>
<td></td>
<td>T,R,W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service</td>
<td>Specifies the service for the entity. Only for use by RTI infrastructure services, not user applications</td>
<td>BE</td>
<td>D,R,W</td>
<td>Y</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----------------------------</td>
<td>--------------------------------------------------------------</td>
<td>-------</td>
<td>------</td>
<td>-----</td>
<td>-------</td>
<td>----------</td>
<td>------</td>
</tr>
<tr>
<td>SystemResourceLimits</td>
<td>Configures process-level resources, independent of particular entities</td>
<td>(Y) BC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TimeBasedFilter</td>
<td>Sets minimum time period before new data for an instance is provided to a DataReader</td>
<td>BE</td>
<td>Y</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TopicQueryDispatch</td>
<td>Configures the ability of a DataWriter to publish historical samples</td>
<td>BC</td>
<td></td>
<td></td>
<td>W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TransferMode</td>
<td>Configures the properties of a Zero Copy DataWriter</td>
<td>(Y) BC</td>
<td></td>
<td></td>
<td>W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TransportBuiltin</td>
<td>Specifies which built-in transports are used</td>
<td>BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TransportMulticast</td>
<td>Sets multicast address and port number on which DataReader will receive data</td>
<td>BC</td>
<td></td>
<td></td>
<td>R</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TransportMulticastMapping</td>
<td>Specifies automatic mapping between list of topic expressions and multicast addresses used by DataReaders to receive data</td>
<td>BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TransportPriority</td>
<td>On OS/transport combinations that understand priority, specifies the priority on a per-DataReader/DataWriter basis</td>
<td>BC</td>
<td></td>
<td></td>
<td>T,R,W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TransportSelection</td>
<td>Selects which transports a DataReader/DataWriter may use to send or receive its data</td>
<td>BC</td>
<td></td>
<td></td>
<td>R,W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TransportPriority</td>
<td>Specifies a subset of transports and a port number that can be used by an Entity to receive data</td>
<td>BC</td>
<td></td>
<td></td>
<td>D,R,W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>TypeConsistencyEnforcement</td>
<td>Sets rule to determine if a type used to publish a given topic is consistent with that used to subscribe to it</td>
<td>BE</td>
<td></td>
<td></td>
<td>R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TypeSupport</td>
<td>Attaches application-specific values to a DataWriter/DataReader that are passed to serialization/deserialization routine of associated data type. Controls if padding bytes are set to 0 during serialization.</td>
<td>(Y) A *</td>
<td></td>
<td></td>
<td>D,R,W</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>UserData [GroupData, TopicData]</td>
<td>Attaches arbitrary application data (a buffer of bytes) to discovery meta-data</td>
<td>A</td>
<td></td>
<td></td>
<td>R,W[P,S; T]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>WireProtocol</td>
<td>Configures properties for the DDS wire protocol (RTPS)</td>
<td>(Y) BC</td>
<td></td>
<td></td>
<td>D</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>WriterDataLifecycle</td>
<td>Controls how a DataWriter handles the lifecycle of the instances that it is registered to manage</td>
<td>A</td>
<td></td>
<td></td>
<td>W</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## QoS Policy List

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Transport (Micro only)</td>
<td>Sets available transports</td>
<td>Y</td>
<td>BC</td>
<td>D.R.W</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>UserTraffic (Micro only)</td>
<td>Configures properties of network traffic caused by exchange of user data</td>
<td>Y</td>
<td>BC</td>
<td>D</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

**Legend:**

- **Chg.** = Changeable
- **RxO** = Request/Offered Semantics
- **Inst.** = Behavior is applied per-instance
- **Ext.** = Extension to DDS standard
- **Micro** = Available in Connext DDS Micro 3.
- See the Connext DDS Micro Release Notes at [https://community.rti.com/documentation](https://community.rti.com/documentation) for the latest information on supported QoS

**Chg.**

- A = Always
- BC = Before Creation
- BE = Before Enable
- * = BC in Connext DDS Micro

**Entities**

- F = DomainParticipantFactory
- D = DomainParticipant
- T = Topic
- P = Publisher
- S = Subscriber
- W = DataWriter
- R = DataReader

**Supported?**

- Y = Yes
- (Y) = Partially Yes
- Blank = No
# QoS Policies per Entity

<table>
<thead>
<tr>
<th>DomainParticipant</th>
<th>Topic</th>
</tr>
</thead>
<tbody>
<tr>
<td>UserData</td>
<td>TopicData</td>
</tr>
<tr>
<td>EntityFactory</td>
<td>Durability</td>
</tr>
<tr>
<td>WireProtocol</td>
<td>DurabilityService</td>
</tr>
<tr>
<td>TransportBuiltin</td>
<td>Deadline</td>
</tr>
<tr>
<td>TransportUnicast</td>
<td>LatencyBudget</td>
</tr>
<tr>
<td>Discovery</td>
<td>Liveliness</td>
</tr>
<tr>
<td>DomainParticipantResourceLimits</td>
<td>Reliability</td>
</tr>
<tr>
<td>Event</td>
<td>DestinationOrder</td>
</tr>
<tr>
<td>ReceiverPool</td>
<td>History</td>
</tr>
<tr>
<td>Database</td>
<td>ResourceLimits</td>
</tr>
<tr>
<td>DiscoveryConfig</td>
<td>TransportPriority</td>
</tr>
<tr>
<td>Property</td>
<td>Lifespan</td>
</tr>
<tr>
<td>EntityName</td>
<td>Ownership</td>
</tr>
<tr>
<td>TransportMulticastMapping</td>
<td>DataRepresentation</td>
</tr>
<tr>
<td>Service</td>
<td>EntityFactory</td>
</tr>
<tr>
<td>TypeSupport</td>
<td>ExclusiveArea</td>
</tr>
<tr>
<td>Transport</td>
<td>EntityName</td>
</tr>
<tr>
<td>(Micro only)</td>
<td></td>
</tr>
<tr>
<td>UserTraffic</td>
<td></td>
</tr>
<tr>
<td>(Micro only)</td>
<td></td>
</tr>
</tbody>
</table>

### Publisher

<table>
<thead>
<tr>
<th>Presentation</th>
<th>EntityFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>AsynchronousPublisher</td>
</tr>
<tr>
<td>Group Data</td>
<td>Exclusive Area</td>
</tr>
<tr>
<td></td>
<td>EntityName</td>
</tr>
</tbody>
</table>

### Subscriber

<table>
<thead>
<tr>
<th>Presentation</th>
<th>EntityFactory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition</td>
<td>ExclusiveArea</td>
</tr>
<tr>
<td>Group Data</td>
<td>EntityName</td>
</tr>
</tbody>
</table>

### DataWriter

<table>
<thead>
<tr>
<th>Durability</th>
<th>DataTag</th>
</tr>
</thead>
<tbody>
<tr>
<td>DurabilityService</td>
<td>DataWriterResourceLimits</td>
</tr>
<tr>
<td>Deadline</td>
<td>DataWriterProtocol</td>
</tr>
<tr>
<td>LatencyBudget</td>
<td>TransportSelection</td>
</tr>
<tr>
<td>Liveliness</td>
<td>TransportUnicast</td>
</tr>
<tr>
<td>Reliability</td>
<td>PublishMode</td>
</tr>
<tr>
<td>DestinationOrder</td>
<td>Property</td>
</tr>
<tr>
<td>History</td>
<td>Service</td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>Batch</td>
</tr>
<tr>
<td>TransportPriority</td>
<td>MultiChannel</td>
</tr>
<tr>
<td>Lifespan</td>
<td>Availability</td>
</tr>
<tr>
<td>UserData</td>
<td>EntityName</td>
</tr>
<tr>
<td>Ownership</td>
<td>TopicQueryDispatch</td>
</tr>
<tr>
<td>OwnershipStrength</td>
<td>TransferMode</td>
</tr>
<tr>
<td>WriterDataLifecycle</td>
<td>TypeSupport</td>
</tr>
<tr>
<td>DataRepresentation</td>
<td>Transport (Micro only)</td>
</tr>
</tbody>
</table>

### DataReader

<table>
<thead>
<tr>
<th>DataTag</th>
<th>Durability</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataReaderResourceLimits</td>
<td>Deadline</td>
</tr>
<tr>
<td>DataReaderProtocol</td>
<td>LatencyBudget</td>
</tr>
<tr>
<td>TransportSelection</td>
<td>Liveliness</td>
</tr>
<tr>
<td>TransportUnicast</td>
<td>Reliability</td>
</tr>
<tr>
<td>TransportMulticast</td>
<td>DestinationOrder</td>
</tr>
<tr>
<td>Property</td>
<td>History</td>
</tr>
<tr>
<td>ResourceLimits</td>
<td>ResourceLimits</td>
</tr>
<tr>
<td>UserData</td>
<td>UserData</td>
</tr>
<tr>
<td>Ownership</td>
<td>Ownership</td>
</tr>
<tr>
<td>TimeBasedFilter</td>
<td>TopicQueryDispatch</td>
</tr>
<tr>
<td>ReaderDataLifecycle</td>
<td>TransferMode</td>
</tr>
<tr>
<td>DataRepresentation</td>
<td>TypeSupport</td>
</tr>
<tr>
<td>TypeConsistencyEnforcement</td>
<td>DataRepresentation</td>
</tr>
</tbody>
</table>

### DomainParticipantFactory

<table>
<thead>
<tr>
<th>EntityFactory</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>SystemResourceLimits</td>
<td>Logging</td>
</tr>
</tbody>
</table>
QoS Policy Descriptions

For more detailed information about the following QoS Policies, see the RTI Connext DDS Core Libraries User's Manual.

AsynchronousPublisher

Configures the mechanism that sends user data in an external middleware thread.

Must be used to send "large" data reliably. "Large" means data that cannot be sent as a single packet by a transport. For example, to send data larger than 63K reliably using UDP/IP, you must configure Connext DDS to send the data using asynchronous publishers.

Can reduce the amount of time spent in the user thread to send data.

For small samples, can reduce bandwidth consumption. Using an asynchronous publisher provides aggregation of samples (collecting many RTPS data samples into 1 network packet) since samples written for the same topic to the same destination are coalesced within the same UDP packet, which reduces bandwidth. For bandwidth reduction, see also Batch QoS. However, for aggregation the preferred mechanism is Batch.

Often used with:

- PublishMode—activates asynchronous publishing, a prerequisite for the use of this policy
- LatencyBudget—controls the scheduling of asynchronously published data onto the network across Data Writers

Partially supported by Connext DDS Micro — See information about FlowControllers in the “Sending Large Data” section of the Connext DDS Micro User’s Manual.

Availability

Configures Collaborative DataWriters and Durable Subscriptions.

For a Collaborative DataWriter, specifies the group of DataWriters expected to collaboratively provide data and the timeouts that control when to allow data to be available that may skip samples.

For a Durable Subscription, configures a set of Durable Subscriptions for a DataWriter.

Supported in release 4.5e, 4.6a and higher.

Batch

Configures mechanism for collecting multiple user data samples to be sent in a single network packet.
This allows you to take advantage of the efficiency of sending larger packets and thus increase effective throughput.

Can dramatically increase the effective throughput for small data packets. Usually, throughput for small packets (data < 2048 bytes) is limited by CPU capacity, not by network bandwidth. Batching many smaller data packets so they are sent in 1 large packet increases network utilization and thus throughput in terms of user data packets per second.

For Reliable topics, batching reduces the Reliable (RTPS) traffic because the Reliable entity is the batch, not the sample.

Although this QoS is not supported by Connext DDS Micro, receiving batched samples is supported and enabled by default in Connext DDS Micro.

**Database**

Configures threads and resource limits that control the internal Connext DDS database.

Essentially a list of lists, RTI uses an internal "database" to store information about entities created locally as well as remote entities of other participants found during the discovery process. This QoS configures how RTI manages its database including how often it cleans up, priority of the database thread, and limits on resources that may be allocated by the database.

You may be interested in modifying the shutdown_timeout and shutdown_cleanup_period parameters to decrease the time it takes to delete a DomainParticipant when your application is shutting down.

**DataReaderProtocol, DataWriterProtocol**

Configures DDS on-the-wire protocol (RTPS).

Connext DDS uses a standard protocol for packet (user- and meta-data) exchange between applications. These QoS policies control configurable portions of the protocol, including configuration of the reliable data delivery mechanism of the protocol on a per DataWriter/DataReader basis.

These parameters control timing and timeouts and give you the ability to tradeoff between speed of data-loss detection and repair, versus the network and CPU bandwidth used to maintain reliability.

It is important to tune the reliability protocol (on a per DataWriter and DataReader basis) to meet the requirements of the end-user application so that data can be sent between DataWriters and DataReaders in an efficient and optimal manner in the presence of data loss. You can also use these QoS policies to control how Connext DDS responds to "slow" reliable DataReaders or ones that disconnect or are otherwise lost.

See the [Reliability QoS policy](https://www.rtiodge.com/dev/dds-qos-policy#Reliability) for more on configuring per-DataReader/DataReader reliability. The [History](https://www.rtiodge.com/dev/dds-qos-policy#History) and [ResourceLimits](https://www.rtiodge.com/dev/dds-qos-policy#ResourceLimits) QoS also play an important role in the reliable protocol.
Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS Professional and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**DataReaderResourceLimits, DataWriterResourceLimits**

Configures how DataReaders/DataWriters allocate and use physical memory for internal resources.

DataReaders must allocate internal structures to handle the maximum number of DataWriters that may connect to it, whether or not a DataReader handles data fragmentation and how many data fragments that it may handle, how many simultaneous, outstanding loans of internal memory holding data samples can be provided to user code, as well as others.

DataWriters must allocate internal structures to handle the simultaneously blocking of threads trying to call write() on the same DataWriter, for the storage used by batched DataWriters, and for filters for content-based filtering DataReaders.

Most of these internal structures start at an initial size and by default will grow as needed by dynamically allocating additional memory. You may set fixed, maximum sizes for these internal structures if you want to bind the amount of memory that can be used by a DataReader/DataWriter. Setting the initial size to the maximum size will prevent Connext DDS from dynamically allocating any memory after theDataReader/DataWriter is created.

Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS Professional and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**DataRepresentation**

Specifies which versions of the Extended Common Data Representation (CDR) (version 1 or version 2) and which data compression setting algorithms are offered and requested for your data.

A DataWriter offers one data representation and one data compression algorithm, which indicate the settings the DataWriter uses to serialize and to compress its data. A DataReader requests one or more such representations, which indicate the CDR versions and compression algorithms the DataReader accepts. If a DataWriter's offered representations are contained within a reader’s sequence of requested representations, then the offer satisfies the request, and the policies are compatible. Otherwise, they are incompatible.

Partially supported by Connext DDS Micro — Connext DDS Micro does not interoperate with DataWriters that send compressed data.

**DataTag**

Associates name/value string pairs with DataWriters and DataReaders that may be used by the Access Control plugin.

This QoS can be used to associate a set of tags in the form of name/value string pairs with a DataReader or DataWriter. This is similar to the Property QoS, except you cannot select whether or not a particular pair
Deadline

should be propagated (included in the built-in topic); instead, data tags are always propagated. The Access Control plugin may use the tags to determine publish and subscribe permissions. You can manipulate the sequence of tags (name/value pairs) with the standard methods available for sequences.

See also:

- [UserData, GroupData, TopicData](#)—these QoS policies also attach discoverable meta-data to entities.
- [DomainParticipantResourceLimits](#)—contains several parameters for configuring the resources associated with the data tags stored in this QoS.

Supported by Connext DDS Micro

**Deadline**

For DataReaders: specifies the maximum expected elapsed time between arriving data samples. For DataWriters: specifies a commitment to publish samples with no greater than this elapsed time between them.

Can be used during system integration to ensure that applications have been coded to meet design specifications.

Can be used at runtime to detect when systems are performing outside of design specifications. Receiving applications can take appropriate actions to prevent total system failure when data is not received in time. For topics on which data is not expected to be periodic, the deadline period should be set to an infinite value.

Loss of Deadline results in loss of ownership for topics with Exclusive Ownership.

Supported by Connext DDS Micro

**DestinationOrder**

Controls how Connext DDS deals with data sent by multiple DataWriters for the same topic.

When multiple DataWriters send data for the same Topic, the order in which data from different DataWriters is received by the applications of different DataReaders may be different. So different DataReaders may not receive the same "last" value when DataWriters stop sending data.

If set to "by reception timestamp":

Data will be delivered by a DataReader in the order in which it was received (which may lead to inconsistent final values).

If set to "by source timestamp":

Data will be delivered by a DataReader in the order in which it was sent. If data arrives on the network with a source timestamp earlier than the source timestamp of the last data delivered, the new data will be dropped. This ordering therefore works best when system clocks are relatively synchronized among writing machines.

Not all data sent by multiple DataWriters may be delivered to a DataReader, and not all DataReaders will see the same data sent by DataWriters. However, all DataReaders will see the same "final" data when DataWriters stop sending data.

Can be used to create systems that have "eventual consistency." Thus intermediate states across multiple applications may be inconsistent, but when DataWriters stop sending changes to the same topic, all applications will end up having the same state.

Partially supported by Connext DDS Micro — Connext DDS Micro supports “by source timestamp” only on DataWriters. Connext DDS Micro DataWriters support “by source timestamp” in case your Connext DDS Professional or other vendor application's DataReaders need it. But a Connext DDS Micro DataReader orders “by reception timestamp” only.

**Discovery**

Configures mechanism used to automatically discover and connect with new remote DomainParticipants.

This QoS specifies the identifiers used by the DomainParticipant to discover other DomainParticipants with which to communicate. This is done via an initial peers list. The middleware will periodically send network packets to these locations, announcing itself to any remote applications that may be present, and will listen for announcements from those applications. This QoS policy also controls the extent of and the transports used for discovery.

Often used with: [DiscoveryConfig] to tune timeliness and reliability related parameters for the discovery process.

Partially supported by Connext DDS Micro — Discovery is configured differently in Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**DiscoveryConfig**

Configures timeliness and reliability settings for discovery.

This QoS policy controls how often discovery data is sent as well as the reliability settings used for discovery topics. The amount of network traffic required by the discovery process can vary widely, based on how your application has configured the middleware's network addressing (e.g., unicast vs. multicast, multicast TTL, etc.), the system’s size, whether all applications are started at the same time or at staggered times, and other factors. Your application can use this policy to make tradeoffs between discovery completion time and network bandwidth utilization. You can also introduce random back-off periods into the discovery process to decrease the probability of network contention when many applications start simultaneously.
Often used with: Discovery – to define the discovery peers.

**DomainParticipantResourceLimits**

Configures how DomainParticipants allocate and use physical memory for internal resources, including maximum sizes of various properties.

This QoS policy sets size limits on variable-length parameters used by the participant and its contained entities. It also controls the initial and maximum sizes of data structures used by the DomainParticipant to store information about locally created and remotely discovered entities (such as DataWriters and DataReaders), as well as parameters used by the internal database to size the hash tables it uses.

Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS Professional and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**Durability**

Specifies whether Connext DDS will store and deliver previously published data to new/late-joining DataReaders.

Durability controls whether or not new DataReaders get data that was written by DataWriters previously and how many samples (via writer_depth) the late-joining DataReaders get. The Durability can vary from not at all (the default) to persistent (stored to disk). This QoS policy helps insulate systems from startup dependencies and can increase system tolerance to failure conditions.

Often used with:

- **DurabilityService** – to configure the external service parameters.

Partially supported by Connext DDS Micro — Only VOLATILE and TRANSIENT-LOCAL values are supported. writer_depth is not supported. Note that the INTRA transport in Connext DDS Micro does not support Durability. Therefore, Durability is not supported in Connext DDS Micro when using INTRA.

**DurabilityService**

Configures external Persistence Service used for DataWriters with PERSISTENCE/TRANSIENT Durability.

When a DataWriter's Durability QoS is set to PERSISTENT_DURABILITY or TRANSIENT_DURABILITY, an external service, RTI Persistence Service, is used to store and possibly forward the data sent by the DataWriter to DataReaders that are created after the data was initially sent. This QoS policy configures certain parameters of RTI Persistence Service when it operates on behalf of the DataWriter, such as how much data to store.

Often used with: Durability – to specify the level of durability for the data.
**EntityFactory**

Controls an Entity’s behavior as a ‘factory’ for other entities, such as whether child entities are created in the enabled state.

This QoS policy is useful to synchronize the initialization of Entities. For example, when a DataReader is created in an enabled state, its existence is immediately propagated for discovery and the DataReader's listener is called as soon as data is received. The initialization process for an application may extend beyond the creation of the DataReader, so you may not want for the DataReader to start to receive or process any data until the initialization process is complete. By creating readers in a disabled state, you can make sure that no data is received until the rest of the application initialization is complete.

Supported by Connext DDS Micro

**EntityName**

Assigns a name to an Entity and associates a DomainParticipant with a Durable Subscription.

This QoS policy assigns a name to a DomainParticipant, Publisher, Subscriber, DataWriter, or DataReader; this name will be sent as part of during discovery. The name will also be visible in RTI tools (except for Publishers and Subscribers) to help you visualize and debug your system. It also assigns a role name, which specifies the Durable Subscription to which the DataReader belongs.

Often used with: Availability – if using Durable Subscriptions.

Partially supported by Connext DDS Micro — EntityName names the participant in Connext DDS Micro, there is no concept of a Durable Subscription. See the Connext DDS Micro API Reference documentation.

**Event**

Configures the internal thread in a DomainParticipant that handles timed events.

In a DomainParticipant, the Event Thread is dedicated to handling all timed events, including checking for timeouts and deadlines, and executing internal and user-defined timeout or exception handling routines/callbacks. An example of a timed event is the reliable heartbeat rate. If this rate is very high, you should use a higher than normal event thread priority.

This QoS policy allows you to configure thread properties such as priority level and stack size. You can also configure the maximum number of events that can be posted to the event thread. By default, a DomainParticipant will dynamically allocate memory as needed for events posted to the event thread. However, by setting a maximum value or setting the initial and maximum value to be the same, you can either bind the amount of memory allocated for the event thread or prevent a DomainParticipant from dynamically allocating memory for the event thread after initialization.
ExclusiveArea

Configures multi-thread concurrency and deadlock prevention capabilities.

An exclusive area (EA) is an abstraction of a multi-thread-safe region. Each entity is protected by one and only one EA, although a single exclusive area may be shared by multiple entities. Conceptually, an EA is a mutex or monitor with additional deadlock protection features.

Note: one only gets into exclusive area violations because entity creation/modifications are done in callbacks. If these operations are one in user threads, exclusive violations cannot occur. Creating shared EA’s is generally not a good practice since it decreases concurrency.

GroupData

Attaches arbitrary application data to discovery meta-data at the publisher/subscriber level.

See: UserData.

History

Controls how much data to store and how stored data is managed for a DataWriter or DataReader.

Controls how Connext DDS manages data sent by a DataWriter or received for a DataReader. Two settings are available: KEEP_ALL or KEEP_LAST with a value (depth). KEEP_ALL does not imply that Connext DDS will store infinite data. How much data can actually be stored (and thus memory allocation) is controlled by the ResourceLimits QoS.

When using the KEEP_LAST setting, the value of depth (number of data samples to keep) applies on a per instance-basis (unique key value) for Topics that are keyed. For example, for a Topic that provides data about the positions of aircraft and using the aircraft ID as a key, a KEEP_LAST setting of 1 will tell Connext DDS to keep the last value for each unique aircraft ID.

The major use of history is to help tune the reliability between DataWriters and DataReaders. You must use KEEP_ALL History for both the DataWriter and DataReader if you want strict-reliability (the DataReader must receive all of the data sent by the DataWriter). Using KEEP_LAST on either side, Connext DDS will only guarantee the reliability of the last N (depth) values sent.

Often used with:

- **Reliability** – using the KEEP_LAST kind for History depth, the level of reliability can be tuned.
- **Durability** – for the DataWriter side to specify how much data need to be stored and sent to new DataReaders who request previously published data using the non-VOLATILE Durability settings.

See also: ResourceLimits
LatencyBudget

Partially supported by Connext DDS Micro — KEEP_ALL is not supported when using the INTRA transport. See the Connext DDS Micro API Reference documentation.

**LatencyBudget**

Suggests how much time is allowed to deliver data.

This is an optional QoS; its implementation and usage may be determined by the implementer.

RTI uses this QoS policy to help schedule data to be sent by an asynchronous thread with a Flow Controller. Data with low latency requirements can be sent ahead of data that is not latency sensitive when using this QoS policy and the asynchronous publishing mode for a DataWriter.

The default value is 0, which implies you want to send with minimum latency.

If using this QoS policy, then all DataWriters whose data is sent by the same Flow Controller must set the value for this QoS policy accordingly. The data of any DataWriter that uses the default value for this QoS policy will always be sent ahead of DataWriters with non-zero values for this QoS policy.

Only used with:

- **PublishMode** – specifically when using DDSASYNCHRONOUS_PUBLISH_MODE_QoS and a flow controller.
- DDSFlowController – specifically the DDS_EDF_FLOW_CONTROLLER_SCHED_POLICY for DDS_FlowControllerProperty_t.

**Lifespan**

Specifies how long Connext DDS should consider data sent by a user application to be valid.

Connext DDS will timestamp all data sent and received. When a finite Lifespan is specified for a DataWriter orDataReader, Connext DDS will check to see how long the data has been stored in the DataWriter’s send queue or the DataReader’s receive queue and remove any data that has exceeded its Lifespan duration.

Connext DDS will always compare times from the same local clock when checking Lifespan. Thus, the Lifespan for data in the DataReader uses the time as stamped by the local clock when the data was received as the start time for the Lifespan calculation.

You can use this QoS policy to limit how much data is stored by Connext DDS. Even if configured for KEEP_ALL History QoS, data may be dropped by Connext DDS due to the use of the Lifespan QoS.

May interact with the Reliability QoS. Due to expired Lifespan, data in the DataWriter that has not yet been acknowledged as have been received in a reliable connection may be dropped and thus never repaired if the initial sample was lost.
Data in the DataReader that has not yet been accessed by the user code may also be removed due to an expired Lifespan.

You can use this QoS policy to ensure applications don’t receive or act on data, commands, or messages that are too old and have "expired."

**Liveliness**

Configures the mechanism that allows DataReaders to detect when matching DataWriters become disconnected or dead.

In a connectionless communications model, the only way a DataReader can know if it is able to receive data from a DataWriter is if it actually receives a message from the DataWriter. This message can either be real data sent by user code, or a message that indicates that the DataWriter is indeed alive and that all elements of the communications link between the DataWriter and DataReader applications is working (network cards, switches, routers, etc.).

By using the Liveliness QoS, the user configures how the DataWriter application maintains liveliness with the DataReader application and how fast the DataReader application is able to detect when the DataWriter is unable to send data to the DataReader (because the sending application has died or perhaps the link between the two applications has broken).

You can use this to ensure that important messages can be received if they are sent. For example, if a command message like “Emergency Stop” is never sent unless the situation is encountered, Liveliness can be used to periodically send a message to test the end-to-end connectivity of the system so that when an “Emergency Stop” message is sent, it will likely be received by all subscribers.

You can use it at run time to detect when systems are performing outside of design specifications. Receiving applications can take appropriate actions in response to disconnected DataWriters.

Partially supported by Connext DDS Micro — The assertions_per_lease_duration parameter is not supported by Connext DDS Micro (it is always assumed to be the default of 3).

**Logging**

Configures the logging feature.

Allows users to configure various aspects associated with logging via XML profile files: verbosity, category, print format, and output file.

**MultiChannel**

Configures DataWriter to send data using different multicast groups (addresses) based on data value.

Data published by a DataWriter can be divided into different “channels” within the same topic. This is useful when the total volume is data is large—perhaps even larger than the capacity of a single network link—and the number of DataReaders is large, but each DataReader is interested in only a subset of the available
data. A “channel” is defined by a filter expression and a set of multicast addresses (groups). When data is sent, it is passed through the filter expressions for all defined channels to see if it will be sent on a particular channel. The filter expressions are applied on the value of the data (contents of the data structure). It is possible for the same data value to be sent on multiple channels. When sent on a channel, Connext DDS will send the data on all of the multicast addresses defined for the channel.

A DataWriter using the MultiChannel QoS policy will ignore the DataReader’s Transport Unicast and Transport Multicast QoS settings. It will only send the data on the multicast addresses set for its channels. DataReaders are expected to subscribe to Content-Filtered Topics and will automatically subscribe to data using the configured multicast addresses of a multi-channel DataWriter for the channels whose filter intersects the Content-Filtered Topic of the DataReader.

Usually, the data of different instances within a topic are grouped together into different channels. Since there are usually more objects than multicast addresses that can be practically used, a single multicast address assigned to a single channel will usually carry the data for more than one instance. So using multi-channel DataWriters will greatly reduce, but probably not eliminate, the undesired delivery and discarding of unwanted data by the subscribing machine.

Users must analyze the data usage across all applications/machines in their system and map out the data topology to create an optimal mapping (channel filters) of objects/instances/independent substreams to minimize network bandwidth usage as well as the CPU usage for dropping unwanted data.

Besides using Content-Filtered Topics to configure the substream of data desired, there is no additional configuration that is needed for DataReaders.

See also: ContentFilteredTopics in the [RTI Connext DDS Core Libraries User's Manual](https://www.rti.com/products/distributed-realtime-apis/dds-core) (not a QoS policy)

**Ownership**

Specifies if a DataReader can receive new samples for an instance of data from multiple DataWriters at the same time.

By default, DataReaders for a given topic can receive data from any matching DataWriter for the same topic—this is the "shared" setting for the Ownership QoS policy. You can also configure an application to use "exclusive" Ownership for a topic so that DataReaders only receive data from one DataWriter at a time. Ownership applies on a per key value (instance) basis for Topics that are keyed. Thus, with exclusive Ownership, DataReaders will only receive data from a single DataWriter for a particular instance (unique value for the key) of a Topic. This implies that a DataReader may receive data from multiple DataWriters as long as the DataWriters are sending data for different instances.

The value of Ownership must be the same for a DataWriter to be connected to a DataReader. Either both sides must be shared or both sides must be exclusive. Mismatched DataWriter and DataWriter pairs are not connected and will never exchange data.
The Ownership QoS policy can also be used to create data channels or Topics designed to be taken over by external applications for testing or maintenance.

Often used with:

- **Liveliness**—to detect when DataWriters are dead and change ownership.
- **Ownership Strength**

Supported by Connext DDS Micro.

**OwnershipStrength**

The OwnershipStrength QoS policy is used to determine which DataWriter is allowed to send data (or updates for instances for keyed Topics) to DataReaders when Ownership is exclusive and there are multiple DataWriters all sending data for the same instance. The DataWriter with the highest value for the OwnershipStrength QoS policy will be considered the owner of the instance of the Topic and whose data is delivered to DataReaders. Data for the instance sent by all other DataWriters with lower OwnershipStrength will be dropped by Connext DDS when received at the subscribing application.

An arbitrary but deterministic method is used to select a single DataWriter among multiple DataWriters with the same highest OwnershipStrength, so all DataReaders will receive the data from the same DataWriter even when the OwnershipStrength is the same value.

When the DataWriter with the highest Ownership strength loses its liveness (as controlled by the Liveliness QoS policy) or misses a deadline (as controlled by the Deadline QoS policy) or whose application quits, dies, or otherwise disconnects, Connext DDS will change ownership of the Topic instance to the DataWriter with the highest OwnershipStrength from the remaining DataWriters.

This QoS policy can help you build systems that have redundant elements to safeguard against component or application failures. When systems have active and hot standby components, the Ownership QoS policy can be used to ensure that data from standby applications are only delivered in the case of the failure of the primary.

Ownership QoS policy can also be used to create data channels or Topics designed to be taken over by external applications for testing or maintenance.

Often used with:

- **Deadline**—to detect when DataWriters have not sent data in time so that a lower strength data is accepted.

Supported by Connext DDS Micro
Partition

Adds additional string identifiers for matching DataWriters and DataReaders for the same Topic.

Specified for Publishers and Subscribers, the Partition QoS policy can be used to add additional identifiers in the form of strings that Connext DDS will used to match DataWriters and DataReaders. Normally, DataWriters are connected (matched) to DataReaders of the same Topic (assuming that their QoS settings are mutually compatible). However, by using the Partition QoS policy, additional criteria is used to decide if a DataWriter’s data is allowed to be sent to a DataReader. Referred to as partitions, one or more strings can be added to the DataWriter’s Publisher or DataReader’s Subscriber parent. When the Partition QoS policy is used, then a DataWriter is only connected (matched) to a DataReader for the same Topic only if their Publisher and Subscribers have a common partition (intersecting partitions).

Direct comparison of partition strings of the Publisher and Subscriber is the nominal way to determine if there is a common partition. However, one can also use strings that contain wildcards (actually regular expressions as defined by the POSIX fnmatch API) as partitions. In that case, as long as one of the wildcard partitions matches one of the concrete partition strings on the other side, then the DataWriter and DataReader are connected.

The set of partitions of a Publisher or Subscriber can be dynamically changed. This can be used to quickly control which DataWriters and which DataReaders are allowed to connect. This facility is useful for creating temporary separation groups among entities that would otherwise be connected to and exchange data each other. Therefore, if sometimes you want an application (or individual DataWriter or DataReader) to be connected to another application (or DataReader or DataWriter, respectively) and at other times you don’t, then you can use the Partition QoS policy to dynamically configure the connection topology without stopping/staring or destroying/re-creating applications or Entities.

Presentation

Controls how Connext DDS presents data received by an application to the DataReaders of the data

Usually DataReaders receive data in the order it was sent by a DataWriter. (When using the Reliability QoS policy, samples which arrive out of order will be buffered until all previous samples arrive.) Once in order, data is presented to the DataReader immediately.

Sometimes you may want a set of data for the same topic to be presented to the receiving DataReader only after ALL of the elements of the set have been received, but not before. You may also want the data to be presented in a different order than it was received. Specifically, for keyed data, you may want Connext DDS to present the data grouped by instance (same key value).

Thus this QoS policy allows you to specify different scopes of presentation, within a topic, across instances of a topic, and across different Topics of a Publisher (although this last option is not currently supported by RTI). It also controls whether or not a set of changes within the scope is delivered at the same time or can be delivered as soon as each element is received.
See also: DestinationOrder—also impacts the order in which samples will be delivered by the middleware to the application.

A Connext DDS Micro DataWriter always announces ordered_access = TRUE and access_scope = TOPIC as part of DataWriter discovery. It is not possible to set this QoS policy in Connext DDS Micro.

Profile

Configures how XML documents containing QoS profiles are loaded by the DomainParticipantFactory.

QoS values for Entities can be configured in QoS profiles defined in XML documents. The XML documents that define QoS profiles can be passed to Connext DDS as a string in XML format or loaded through XML files found on a file system.

The Profile QoS policy for the DomainParticipantFactory configures Connext DDS to load XML documents via 3 independent mechanisms that may be used together or individually disabled.

Environment variable – the value of the environment variable NDDS_QOS_PROFILES can be used to set the locations of files on the file system that contain XML documents to be loaded automatically by Connext DDS. You can also set a XML-formatted string as a value for the environment variable. You can disable the use of the environment variable using the Profile QoS policy.

Default XML files – there are default locations where Connext DDS will look for XML files to load QoS profiles. These include the current working directory from where an application is started and a file in the distribution directory for Connext DDS. You may disable any or all of these default locations using the Profile QoS policy.

The locations of files as well as strings containing XML-formatted QoS profiles can be specified directly via the Profile QoS policy.

Property

Specifies name/value pairs used to configure parameters not exposed via formal QoS policies

The Property QoS policy allows name/value pairs (in the form of strings) to be attached to the QoS of different entities. These strings are often used to add configuration parameters to an entity without having to change formal QoS policies. Behaviors configured by means of this policy are often not known a priori (for example, dynamically loaded network transport plug-ins) or are provided on a provisional basis. You may also add your own name/value pairs to the Property QoS policy of an Entity and direct Connext DDS to propagate this information with the Entity’s discovery information so it can be accessed in other applications. This allows you to add meta-information about an Entity for application-specific uses.

- Used to configure the property of built-in-ports before DomainParticipant creation.
- Used to specify dynamic loading of extension transports (such as RTI Secure WAN Transport).
- Used to specify multiple instances of the built-in transports.
- Allows full pluggable transport configuration for non-C/C++ languages (Java, C#, etc.).
- Used to select a clock.

Configures Durable Writer History and Durable Reader State for high reliability in the face of component or system failures (see the chapter on "Mechanisms for Achieving Information Durability and Persistence" in the RTI Connext DDS Core Libraries User's Manual).

Allows you to attach additional information to entities—such as for identification, authentication, and authorization.

Often used with:

- **TransportBuiltin, TransportSelection, TransportUnicast, TransportMulticast**—configures aspects of built-in and dynamically loaded transports.
- **DomainParticipantResourceLimits**—configures maximum size of discoverable properties.
- **Availability, Durability, DurabilityService**—also relevant to highly reliable and available systems.

See also: **UserData, GroupData, TopicData, DataTag**—these QoS policies also attach discoverable meta-data to entities.

Supported by Connext DDS Micro only when also using RTI Connext DDS Secure. This QoS is always supported in Connext DDS Professional, but in Connext DDS Micro it is supported only if you are using Connext DDS Secure. (Connext DDS Micro uses this QoS only for Connext DDS Secure.)

**PublishMode**

Specifies the mechanism that sends user data in an external middleware thread

By default, data is sent in the context of the user thread that calls write()—this is typically the lowest-latency approach. However, there are times when you may want an internal middleware thread to send the data asynchronously instead. Thus when the user thread calls write(), the data is only serialized and copied to an internal buffer. A thread owned by the Publisher is responsible for sending the data when it is scheduled. This QoS policy configures this behavior. It may be SYNCHRONOUS (default) or ASYNCHRONOUS. You can also use this QoS policy to select a Flow Controller to "shape" the flow of data onto the network.

"Shaping" a data flow usually means limiting the maximum data rates that Connext DDS will use to send data for a DataWriter. The flow controller will buffer data sent by the DataWriter faster than the maximum rate configured for the Flow Controller, and then only send the excess data when the send rate drops below the maximum rate.

The Asynchronous publish mode must also be used to send large data with RELIABLE reliability. Large data is any data that exceeds the maximum message (packet) size that a transport plugin is configured to send—between 9 and 64 KB in the case of UDP transports, depending on the configuration.
Note that a single thread will be used to send data for all of the asynchronous DataWriter’s of a Publisher. If you want to have different threads to send data asynchronously, you must use multiple Publishers.

Often used with:

- **LatencyBudget**—controls the scheduling of asynchronously published data onto the network across DataWriters.
- **Property**—configures maximum size of available transport plug-ins.
- **AsynchronousPublisher**—lower-level settings pertaining to asynchronous publication, like thread priority and options.

See also: Flow Controllers (must be configured via the API, see the [RTI Connext DDS Core Libraries User's Manual](#))

Supported by Connext DDS Micro

**ReaderDataLifecycle**

Controls how a DataReader manages the lifecycle of the data that it has received.

When a DataReader receives data, it caches that data for your application. Your application may either take the data from the cache or leave it there. This QoS policy controls when the middleware will automatically remove the stored data for a particular instance of the Topic from the cache (such that the data is no longer accessible by the application) when it detects that there are no more DataWriters alive for a specific instance or when a DataWriter has disposed the instance. For Topics without keys, this applies when there are no DataWriters alive for the Topic, or if the Topic has been disposed by a DataWriter.

You can use this QoS policy to specify a delay in which the data removal will occur when the middleware detects one of the two states described earlier. By default, the delays are specified to be infinite, which configures Connext DDS not to remove any data from the DataReader’s cache.

**ReceiverPool**

Configures threads used to receive and process data from transports (for example, UDP sockets).

Configures the properties of the “receive” threads that the middleware uses to receive and process data from its installed transports. This includes setting thread properties, such as priority level and stack size for these threads, as well as the size of the buffer used to store packets received from a transport. This buffer size limits the largest single packet of data that a DomainParticipant will accept from a transport. For many applications, the value 65,536 (64 KB) is a good choice, as this is the largest packet that can be sent/received via UDP.

Note that Connext DDS does not currently share threads among the different transport plugins. Instead, each transport plugin is automatically allocated a single thread by default; depending on the transport plugin, it may request a separate thread for each receive port.
Often used with:

- **Property**—configures the maximum message size of individual transports
- **TransportUnicast, TransportMulticast**— RTI uses a separate receive thread per port per transport plug-in by default. To force Connext DDS to use a separate thread to process the data for a DataReader, set a unique port in the DataReader’s Transport Unicast or Transport Multicast QoS policy.

## Reliability

Enables reliability protocol for a DataWriter/DataReader connection.

This QoS policy turns on the RTPS reliability protocol between those DataWriters and DataReaders that set this QoS policy to the RELIABLE value. When the reliability protocol is used, Connext DDS will attempt to repair samples that were not successfully received by reliable DataReaders.

By itself, the Reliability policy does not set the level of reliability for a connection between a DataWriter and DataReader. Instead, the level of reliability is controlled in conjunction with other policies, such as **History** and **ResourceLimits**, to determine which data remains relevant and therefore eligible for repair. For example, as a tradeoff for less memory, CPU, and network usage, you can choose a reduced level of reliability where only the last N values are guaranteed to be delivered reliably to DataReaders (N is configurable). With the reduced level of reliability, there are no guarantees that data sent before the last N are received; only the last N data packets are monitored and repaired if necessary.

A connection between a DataWriter and DataReader may be configured for BEST_EFFORT Reliability, which means Connext DDS will not use any resources to monitor or guarantee that the data sent by a DataWriter is received by a DataReader. For some use cases, such as the periodic update of sensor values to a GUI displaying values to a person, "best effort" delivery is often good enough. It is the fastest, most efficient, and least resource-intensive (CPU and network bandwidth) method of getting the newest/latest value for a topic from DataWriters to DataReaders. However, there is no guarantee that the data sent will be received. It may be lost due to a variety of factors, including data loss by the physical transport such as wireless RF or even Ethernet.

For some data streams (topics), you need to be assured that all data from the DataWriter is received reliably by the DataReaders. Connext DDS will ensure data was received and repair any lost data by resending a copy as many times as needed until the DataReader receives it. This level of reliability is considered “strictly reliable.” For strict reliability, use the KEEP_ALL setting for the **History** QoS policy for both the DataWriter and DataReader along with the RELIABLE setting for the Reliability QoS policy.

RELIABLE DataWriters may send data to both BEST_EFFORT and RELIABLE DataReaders, but RELIABLE DataReaders can only receive data from RELIABLE DataWriters.

The RTPS reliability protocol is highly configurable using the **DataWriterProtocol** and **DataReaderProtocol** QoS policy. **Most users will need to configure the reliability protocol to meet their real-time requirements** since the default configuration is not designed for speedy packet-loss detection and repair. In
the examples directory of the Connext DDS installation, you should find an example QoS configuration for the reliability protocol that is better suited for many applications.

To maintain reliability in the face of component or system failures, you can use RTI Persistence Service to store sent data on other nodes or even to disk elsewhere on the network using the Durability and DurabilityService QoS policies.

Often used with:

- **History**—governs how many previously written samples remain relevant; set history kind to KEEP_ALL for TCP-like reliability over connectionless transports like UDP
- **Lifespan**—governs how long previously written samples remain relevant; data with an expired lifespan will not be repaired even if lost
- **ResourceLimits**—governs the total sizes of DataWriter sample caches used to maintain commitments if a finite reliability send window has not been set (see DataWriterProtocol)
- **Availability, Durability, DurabilityService**—configure additional data sources to maintain reliable delivery in the face of partial or total system failure
- **DataWriterProtocol**—a variety of parameters for controlling the RTPS reliability protocol for a reliable DataWriter, including the size of the reliability “send window” and the rate at which heartbeats are sent. This also controls whether reliability will be maintained only by means of negative acknowledgments, or by a combination of positive and negative acknowledgments (the default).
- **DataReaderProtocol**—includes corresponding parameters for controlling the reliability protocol for the reliableDataReader. Among other things, this controls whether reliability will be maintained only by means of negative acknowledgments, or by a combination of positive and negative acknowledgments (the default).

Partially supported by Connext DDS Micro — The acknowledgment_kind parameter is not supported by Connext DDS Micro. The INTRA transport in Connext DDS Micro does not support reliability—that is, lost data is not retransmitted.

**ResourceLimits**

Controls amount of physical memory is allocated for middleware entities; if dynamic allocations are allowed and how they occur; and memory usage among different instance values for keyed topics.

Configures the amount of memory a DataWriter or DataReader may allocate to store data in a local cache (also referred to as send or receive queues, respectively). The max_samples parameter in this policy also has a role in throttling the send rate of reliable DataWriters, although starting in version 4.5, using the “send window” properties of the DataWriterProtocol QoS policy is the recommended way to configure this behavior.
This QoS policy can limit how much system memory can be allocated by the middleware. For embedded real-time systems and safety-critical systems, predetermination of maximum memory usage is often required. In addition, dynamic memory allocation may introduce non-deterministic latencies in time-critical paths. By setting the initial and maximum settings to the same values for different parameters of this QoS policy, you can be assured that an entity will not dynamically allocate more memory after its initialization phase.

Often used with:

- **History**—configures the window of relevant previously written samples, which must fit within the limits set by the ResourceLimits policy.
- **Reliability**—the physical data storage limits of the DataWriter and DataReader caches can affect the performance of the protocol for reliable connections.
- See also: DataWriterProtocol—recommended for taking over the role of reliability throttling from the ResourceLimits policy as of version 4.5.

Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**Service**

Specifies the type of service for the entity.

This policy is intended for use by RTI infrastructure services and should not be modified by user applications.

**SystemResourceLimits**

Configures DomainParticipant-independent resources.

Within a single process (or address space for some supported real-time operating systems), applications may create and use multiple DomainParticipants. This QoS policy governs resources at the process level, across DomainParticipants.

This QoS policy places an effective limit on the number of DomainParticipants that can be created in a single process. You may need to modify this QoS policy if you need to create more than 10 DomainParticipants in a process.

Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS Professional and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.
TimeBasedFilter

Sets a minimum time period before new data for an instance is provided to a DataReader, excess data sent faster than the period set are not sent or otherwise discarded.

DataWriters may send data faster than needed by a DataReader. For example, a DataReader of sensor data that is displayed to a human in a GUI application often does not need data updates faster than a human can reasonably perceive changes in data values. The period between updates is often measured in tenths (0.1) of a second up to several seconds. However, a sensor-information DataWriter may have DataReaders that are processing the sensor information to control parts of the system and thus need new data updates with periods of hundredths (0.01) or thousandths (0.001) of a second. This policy allows you to optimize resource usage (CPU and possibly network bandwidth) by only delivering the required amount of data to different DataReaders and filtering out samples that arrive faster than a rate you specify (in terms of a period of time between data arrival).

You can use this QoS policy reduce the amount of data received by a DataReader in order to avoid overwhelming a potentially slow consumer. You can also use it to reduce the amount of data placed on the network by a DataWriter, to avoid overwhelming network resources.

Different DataReaders for the same Topic are allowed to set their own time-based filters, so that data published faster than the period set by a DataReader will be dropped by the middleware and not delivered to the DataReader—without perturbing other DataReaders.

This time-based data-filtering does not depend on any DataWriter settings. However, it also does not force the DataWriter to send at a specific rate. How fast a DataWriter updates new data is entirely under the control of the user code.

See also: ContentFilteredTopics (Not a QoS policy but a feature in which you can create DataReaders that receive data that are filtered based on the value of the data in addition to time. See the RTI Connext DDS Core Libraries User's Manual.)

TopicData

Attaches arbitrary application data to discovery meta-data at the topic level.

See: UserData.

TopicQueryDispatch

Configures the ability of a DataWriter to publish samples in response to a TopicQuery.

Enables a DataWriter to publish historical samples upon reception of a TopicQuery and controls how often they are published. Since a TopicQuery selects previously written samples, the DataWriter must have a non-VOLATILE DurabilityQosPolicy::kind. The ReliabilityQosPolicy::kind must be set to RELIABLE.

A TopicQuery may select multiple samples at once. The writer will publish them periodically, independently from newly written samples. TopicQueryDispatchQosPolicy::publication_period configures the
frequency. TopicQueryDispatchQosPolicy::samples_per_period configures the maximum number of samples to publish each period. If the DataWriter blocks during the publication of one of these samples, it will try again the next period. All DataWriters that belong to a single Publisher and enable TopicQueries share the same event thread, but each DataWriter schedules separate events. To configure that thread, see AsynchronousPublisherQosPolicy::topic_query_publishing_thread. If the DataWriter is dispatching more than one TopicQuery at the same time, the configuration of this periodic event applies to all of them.

**TransferMode**

Configures the properties of a Zero Copy DataWriter.

This QoS policy specifies the properties of a Zero Copy DataWriter. It controls whether or not the DataWriter supports data consistency checks. By default, this QoS enables support for data consistency checks.

Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS Professional and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**TransportMulticast**

Specifies the multicast address on which a DataReader is to receive its data.

By default, DataWriters will send individually addressed packets to each DataReader that subscribes to the topic of the DataWriter; this is known as unicast delivery. One copy of the data is sent for each
TransportBuiltin

Property

Selects TransportSelection can and network Some value Tells TransportPriority must be used Specifies DataReader. This configuration is the simplest to configure and the easiest to manage. However, the network bandwidth used by a DataWriter increases linearly with the number of DataReaders. Multicast addressing (on UDP/IP transports) allows a DataWriter to send a single network packet that will be received by many DataReaders. Thus the network bandwidth usage will be nearly constant independent of the number of DataReaders.

Coordinating the multicast address specified by DataReaders can help optimize network bandwidth usage in systems where there are multiple DataReaders for the same Topic. Note that you can also specify a port number as well as a subset of the available transports with which to receive the multicast data. These settings—multicast address, port number, and transports—do not need to be set uniformly for all DataReaders of the same Topic. A DataWriter will automatically detect which DataReaders can be reached at the same network location and which other DataReaders need to be addressed separately.

See also:

- **TransportBuiltin**—Identifies the available built-in transports
- **Property**—Used to install external (that is, not builtin) transports

**TransportMulticastMapping**

Specifies the automatic mapping between a list of topic expressions and multicast addresses that can be used by a DataReader to receive data for a specific topic.

This QoS policy provides an easy way to configure and assign multicast addresses to DataReaders based on the names of their associated Topics. You can specify an automatic mapping between a list of topic expressions and a multicast address that can be used by the DataReader to receive the data. To have a DataReader use this QoS policy to select its multicast address, its own **TransportMulticast** QoS Policy must be set accordingly.

**TransportPriority**

Tells Connext DDS that the data being sent has a different "priority" than other data. Provides a priority value to the underlying transport protocol, for those that can use it.

Some transport protocols have a concept of user-settable “priorities” that may be used by operating system network stacks and switching hardware in between the publisher and the subscriber. For such transports and on supported OSs, Connext DDS will pass this value to the transport for its use. For other transports and other OSs, the middleware will ignore this value. For more information about whether Connext DDS can preserve this value for your transports and OSs, please consult the Platform Notes.

**TransportSelection**

Selects which transports a DataWriter or DataReader may use to send or receive its data.
This QoS policy is used in the advanced case where multiple transports have been installed and only a subset of them are to be used with a particular DataReader or DataWriter. For example, suppose that transports for UDP/IP, shared memory, Infiniband, and VME transports are available, and a DataReader should only receive data over Infiniband.

See also:

- **TransportBuiltin**—Identifies the available built-in transports
- **Property**—Used to install external (that is, not built in) transports

**TransportUnicast**

Specifies a subset of transports and a port number that can be used by an Entity to receive data.

Connext DDS may send data to a variety of Entities, not just DataReaders. For example, reliable DataWriters may receive ACK/NACK packets from reliable DataReaders. During discovery, each Entity announces to remote applications a list of (up to 4) unicast addresses to which the remote application should send data (either user data packets or reliable protocol meta-data such as ACK/NACKs and heartbeats).

By default, the list of addresses is populated automatically with values obtained from the enabled transport plug-ins that are allowed to be used by the Entity (see the **TransportBuiltin** and **TransportSelection** QoS policies). The associated ports are automatically determined. This default behavior is appropriate for most applications.

Use this QoS policy to manually select a subset of the available receive addresses for an Entity or to specify a non-default receive port.

See also:

- **TransportBuiltin**—Specifies the available built-in transports at the DomainParticipant level
- **Property**—Used to install additional external transports at the DomainParticipant level
- **TransportSelection**—Specifies a subset of the available built-in and external transports available for a particular DataWriter or DataReader

**TypeConsistencyEnforcement**

Defines rules that determine whether the type used to publish a given topic is consistent with that used to subscribe to it.

This policy specifies a type consistency kind, either DISALLOW_TYPE_COERCION or DDS_ALLOW_TYPE_COERCION (default). When Connext DDS is introspecting the built-in topic data declaration of a remote DataReader in order to determine whether it can match with a local DataWriter, if
it observes that no TypeConsistencyEnforcementQosPolicy value is provided, it assumes a kind of DISALLOW_TYPE_COERCION.

**TypeSupport**

Attaches application-specific values to a DataWriter/DataReader, which are passed to the serialization/deserialization routine of the associated data type. Also controls if padding bytes are set to 0 during serialization.

You can modify the rtiddsgen-generated code so that the de/serialization routines act differently depending on the information passed in via the object pointer.

RTI generally recommends that you treat generated source files as compiler outputs (analogous to object files) and that you do not modify them. RTI cannot support user changes to generated source files. Furthermore, such changes would make upgrading to newer versions of Connext DDS more difficult, as this generated code is considered to be a part of the middleware implementation and consequently does change from version to version. The plugin_data parameter in this QoS policy should be considered a back door, only to be used after careful design consideration, testing, and consultation with your RTI representative.

Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS Professional and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**UserData [GroupData, TopicData]**

Attaches arbitrary application data (a buffer of bytes) to discovery meta-data.

These QoS policies attach extra data that you want propagated at discovery time. This extra data is interpreted by the end user application and is not used by the middleware itself.

- UserData attaches discoverable meta-data at the DataWriter/DataReader level.
- GroupData attaches discoverable meta-data at the Publisher/Subscriber level.
- TopicData attaches discoverable meta-data at the Topic level.

Use cases for these policies include application-to-application identification, authentication, authorization, and encryption purposes. For example, applications can use these policies to send security certificates to each other for RSA-type security.

See also:

- [DomainParticipantResourceLimits](#) – Sets the maximum length of the extra data that can be attached (default 256 bytes).
- [Property](#)—an alternative means of attaching discoverable meta-data to entities.
- [DataTag](#)—an alternative means of attaching discoverable meta-data to entities.
**WireProtocol**

Configures the DDS on-the-wire protocol (RTPS).

Configures some global RTPS properties. The DataWriterProtocol and DataReaderProtocol QoS policies configure RTPS and reliability properties on a per DataWriter or DataReader basis.

Partially supported by Connext DDS Micro — The parameters in this QoS differ between Connext DDS Professional and Connext DDS Micro. See the Connext DDS Micro API Reference documentation.

**WriterDataLifecycle**

Controls how a DataWriter handles the lifecycle of the instances that it is registered to manage.

The behavior controlled by this QoS policy applies on a per instance (key) basis for keyed Topics, so that when a DataWriter unregisters an instance, Connext DDS can automatically also dispose that instance. This is the default behavior.

In cases where the ownership of a Topic is exclusive (see the Ownership QoS policy), DataWriters may want to relinquish ownership of a particular instance of the Topic to allow other DataWriters to send updates for the value of that instance regardless of how the OwnershipStrength QoS policy is set. In that case, you may only want a DataWriter to unregister an instance without disposing the instance. Disposing an instance is a statement that an instance no longer exists. User applications may be coded to trigger on the disposal of instances, thus the ability to unregister without disposing may be useful to properly maintain the semantic of disposal.

**Transport (Micro only)**

Sets available transports (used by Connext DDS Micro only).

A sequence of transport identifiers that defines the set of transports available for use by the associated DDS entity.

**UserTraffic (Micro only)**

Configures properties of network traffic caused by exchange of user data (used by Connext DDS Micro only).

Configures the transports enabled to transmit and receive user data traffic.

© 2021 Real-Time Innovations, Inc.