RTI Message Service

Configuration and Operation Manual

Version 5.0

Your systems. Working as one.

Technical Support
Real-Time Innovations, Inc.
232 E. Java Drive
Sunnyvale, CA 94089

Phone:	(408) 990-7444
Email:	support@rti.com
Website:	https://support.rti.com/

Contents

Chapter 1 Welcome to RTI Message Service

Welcome to *RTI® Message Service*, the highest-performing JMS-compliant messaging system in the world. *RTI Message Service* makes it easy to develop, deploy and maintain distributed applications. Its core messaging technology has been proven in hundreds of unique designs for life- and mission-critical applications across a variety of industries, providing

- ❏ ultra-low latency and extremely high throughput
- ❏ with industry-leading latency determinism
- ❏ across heterogeneous systems spanning thousands of applications.

Its extensive set of real-time quality-of-service parameters allows you to fine-tune your application to meet a wide range of timeliness, reliability, fault-tolerance, and resource usage-related goals.

This chapter introduces the basic concepts within the middleware and summarizes how *RTI Message Service* addresses the needs of high-performance systems. It also describes the documentation resources available to you and provides a road map for navigating them. Specifically, this chapter includes:

1.1 Benefits of RTI Message Service

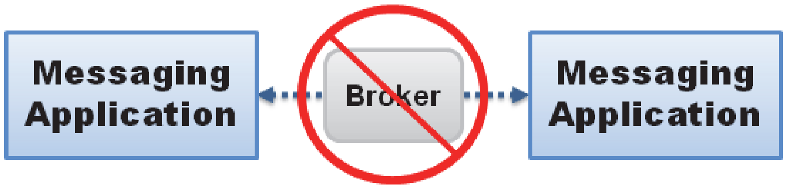
RTI Message Service is publish/subscribe networking middleware for high-performance distributed applications. It implements the Java Message Service (JMS) specification, but it is not just another MOM (message-oriented middleware). Its unique peer-to-peer architecture and targeted high-performance and real-time capabilities extend the specification to provide unmatched value.

1.1.1 Reduced Risk Through Industry-Leading Performance and Availability

RTI Message Service provides industry-leading performance, whether measured in terms of latency, throughput, or real-time determinism. One contributor to this superior performance is RTI's unique architecture, which is entirely peer-to-peer.

Traditional messaging middleware implementations require dedicated servers to broker message flows, crippling application performance, increasing latency, and introducing time non-determinism. These brokers increase system administration costs and can represent single points of failure within a distributed application, putting data reliability and availability at risk.

RTI eliminates broker overhead by allowing messages to flow directly from a publisher to each of its subscribers in a strictly peer-to-peer fashion. At the same time, it provides a variety of powerful capabilities to ensure high availability.



Traditional message-oriented middleware implementations require a broker to forward every message, increasing latency and decreasing determinism and fault tolerance. RTI's unique peer-to-peer architecture eliminates bottlenecks and single points of failure.

Redundancy and high availability can optionally be layered onto the peer-to-peer data fabric by transparently inserting instances of *RTI Persistence Service*. These instances can distribute the load across topics and can also be arbitrarily redundant to provide the level of data availability your application requires. See Chapter 7, "Scalable High-Performance Applications: Durability and Persistence for High Availability," in the User's Manual for more information about this capability.

Publishers and subscribers can enter and leave the network at any time, and the middleware will connect and disconnect them automatically. *RTI Message Service* provides fine-grained control over fail-over among publishers, as well as detailed status notifications to allow applications to detect missed delivery deadlines, dropped connections, and other potential failure conditions. See Chapter 6, "Fault Tolerance," in the Configuration and Operation Manual for more information about these capabilities.

1.1.2 Reduced Cost through Ease of Use and Simplified Deployment

- ❑ **Increased developer productivity**—Easy-to-use, well-understood JMS APIs get developers productive quickly. (Take an opportunity to go through the tutorial in the *Getting Started Guide* if you haven't already.) Outside of the product documentation itself, a wide array of third-party JMS resources exist on the web and on the shelves of your local book store.
- ❑ **Simplified deployment**—Because *RTI Message Service* consists only of dynamic libraries, you don't need to configure or manage server machines or processes. That translates into faster turnaround and lower overhead for your team.
- ❑ **Reduced hardware costs**—Some traditional messaging products require you to purchase specialized acceleration hardware in order to achieve high performance. The extreme efficiency and reduced overhead of RTI's implementation, on the other hand, allows you to see strong performance even on commodity hardware.

1.1.3 Unmatched Power and Flexibility to Meet Unique Requirements

When you need it, RTI provides a high degree of fine-grained, low-level control over the operation of the middleware, including, but not limited to:

- ❑ The volume of meta-traffic sent to assure reliability.
- ❑ The frequencies and timeouts associated with all events within the middleware.
- ❑ The amount of memory consumed, including the policies under which additional memory may be allocated by the middleware.

These quality-of-service (QoS) policies can be specified in configuration files so that they can be tested and validated independently of the application logic. When they are not specified, the middleware will use default values chosen to provide good performance for a wide range of applications.

For specific information about the parameters available to you, consult the Configuration and Operation Manual.

1.1.4 Interoperability with OMG Data Distribution Service-Based Systems

The Data Distribution Service (DDS) specification from the Object Management Group (OMG) has become the standard for real-time data distribution and publish/subscribe messaging for high performance real-time systems, especially in the aerospace and defense industries. *RTI Message Service* is the only JMS implementation to directly interoperate at the wire-protocol level with *RTI Data Distribution Service*, the leading DDS implementation.

RTI Data Distribution Service is available not only in Java but also in several other managed and unmanaged languages. It is supported on a wide variety of platforms, including embedded hardware running real-time operating systems. For more information, consult your RTI account representative. If you are already an *RTI Data Distribution Service* user, and are interested in DDS/JMS interoperability, consult the Interoperability Guide that accompanies this documentation.

1.2 Features of RTI Message Service

Under the hood, *RTI Message Service* goes beyond the basic JMS publish-subscribe model to target the needs of applications with high-performance, real-time, and/or low-overhead requirements and provide the following:

❑ **Peer-to-peer publish-subscribe communications** Simplifies distributed application programming and provides time-critical data flow with minimal latency.

- Clear semantics for managing multiple sources of the same data.
- Efficient data transfer, customizable Quality of Service, and error notification.
- Guaranteed periodic messages, with minimum and maximum rates set by subscriptions, including notifications when applications fail to meet their deadlines.
- Synchronous or asynchronous message delivery to allow applications control over the degree of concurrency.
- Ability to send the same message to multiple subscribers efficiently, including support for reliable multicast with customizable levels of positive and negative message acknowledgement.

- ❑ **Reliable messaging**—Enables subscribing applications to not only specify reliable delivery of messages, but to customize the degree of reliability required. Data flows can be configured for (1) guaranteed delivery at any cost, at one extreme, (2) the lowest possible latency and highest possible determinism, even if it means that some messages will be lost, at the other extreme, or (3) many points in between.
- ❑ **Multiple communication networks**—Multiple independent communication networks (*domains*), each using *RTI Message Service,* can be used over the same physical network to isolate unrelated systems and subsystems. Individual applications can be configured to participate in one or multiple domains.
- ❑ **Symmetric architecture**—Makes your application robust:
    - No central server or privileged nodes, so the system is robust to application and/or node failures.
    - Topics, subscriptions, and publications can be dynamically added and removed from the system at any time.

Multiple network transports—*RTI Message Service* includes support for UDP/IP (v4 and v6)—including, for example, Ethernet, wireless, and Infiniband networks—and shared memory transports. It also includes the ability to dynamically plug in support for additional network transports and route messages over them. It can optionally be configured to operate over a variety of transport mechanisms, including backplanes, switched fabrics, and other networking technologies.

Multi-platform and heterogeneous system support—Applications based on *RTI Message Service* can communicate transparently with each other regardless of the underlying operating system or hardware. Consult the Release Notes to see which platforms are supported in this release.

Vendor neutrality and standards compliance—The *RTI Message Service* API complies with the JMS specification. Unlike other JMS implementations, it also supports a wire protocol that is open and standards-based: the Real-Time Publish/Subscribe (RTPS) protocol specification from the Object Management Group (OMG), which extends the International Engineering Consortium's (IEC's) publicly available RTPS specification. This protocol also enables interoperability between *RTI Message Service* and *RTI Data Distribution Service* and between various DDS implementations. See Interoperability with OMG Data Distribution Service-Based Systems (Section 1.1.4).

1.3 JMS Conformance

RTI Message Service is a high-performance messaging platform for demanding applications, including applications with real-time requirements. Not all portions of the JMS specification are relevant or appropriate for this domain, and some required features are not included in the specification. For more information about JMS conformance, including both limitations and significant extensions, see Appendix A, "JMS Conformance," in the User's Manual.

1.4 Understanding and Navigating the Documentation

To get you from your download to running software as quickly as possible, we have divided this documentation into several parts.

- ❏ Release Notes—Provides system-level requirements and other platform-specific information about the product. *Those responsible for installing RTI Message Service should read this document first.*
- ❏ Getting Started Guide—Describes how to download and install *RTI Message Service*. It also lays out the core value and concepts behind the product and takes you step-by-step through the creation of a simple example application. *Developers should read this document first.*
- ❏ User's Manual—Describes the features of the product, their purpose and value, and how to use them. It is aimed at developers who are responsible for implementing the functional requirements of a distributed system, and is organized around the structure of the JMS APIs and certain common high-level scenarios.
- ❏ Configuration and Operation Manual—Provides lower-level, more in-depth configuration information and focuses on system-level concerns. It is aimed at engineers who are responsible for configuring, optimizing, and administering *RTI Message Service*-based distributed systems.

Many readers will also want to consult additional documentation available online. In particular, RTI recommends the following:

- ❏ **RTI Self-Service Portal**—http://www.rti.com/support. Select the **Find Solution** link to see sample code, general information on *RTI Message Service*, performance information, troubleshooting tips, and other technical details.

- ❑ **RTI Example Performance Test**—This recommended download includes example code and configuration files for testing and optimizing the performance of a simple *RTI Message Service*-based application on your system. The program will test both throughput and latency under a wide variety of middleware configurations. It also includes documentation on tuning the middleware and the underlying operating system.

  To download this test, first log into your self-service support portal as described above. Click **Find Solution** in the menu bar at the top of the page then click **Performance** under **All Solutions** in the resulting page. Finally, click on or search for **Example Performance Test** to download the test.

  You can also review the data from several performance benchmarks here: http://www.rti.com/products/jms/latency-throughput-benchmarks.html.

- ❑ **Java Message Service (JMS) API Documentation**—*RTI Message Service* APIs are compliant with the JMS specification. This specification is a part of the broader Java Enterprise Edition (Java EE) product from Sun Microsystems; Java EE 5 is documented at http://java.sun.com/javaee/5/docs/api/. In particular, see the **javax.jms** package.

- ❑ **Java Standard Edition API Documentation**—Java EE is an extension to, and relies on types imported from, the Java Standard Edition (Java SE) product. Java SE 6 is documented online at http://java.sun.com/javase/6/docs/api/.

- ❑ **Whitepapers and other articles** are available from http://www.rti.com/resources/.

Chapter 2 Debugging the Connection

Eventually, you're likely to encounter a configuration issue or other problem that you need to debug. This chapter describes some of the tools at your disposal.

This chapter includes the following sections:

2.1 Logging Configuration

Class: com.rti.management.Logger

RTI Message Service logs extensive information about its own operation. You can customize what kind of information is logged by using the **Logger** class.

2.1.1 Log Verbosity

Enumeration: com.rti.management.Logger.Verbosity

Method: static Logger.Verbosity getVerbosity()

Method: static void setVerbosity(Logger.Verbosity verbosity)

By default, the middleware only displays error messages; lower-verbosity messages are suppressed. You can change the logging verbosity at any time. The logging levels are identified by constants in the nested **Logger.Verbosity** enumeration:

SILENT	No messages will be logged. This is the minimum level of verbosity.
ERROR	Only error messages will be logged. *This is the default level of verbosity.*
WARNING	Error messages will be logged. The middleware will also log information about situations that *may* represent problems. For example, some configurations may function in limited circumstances, but perhaps not in the way you intended.
STATUS_LOCAL	The middleware will also log tracing information pertaining to the operation of local objects.
STATUS_REMOTE	The middleware will also log tracing information pertaining to the operation of remote objects.
STATUS_ALL	The middleware will display extensive tracing information.

In very-demanding applications, especially those requiring a high degree of determinism, extremely verbose logging can impact performance. RTI recommends that you leave your verbosity set to **ERROR** or **WARNING** unless you are trying to debug a specific problem.

2.1.2 Logging by Functional Categories

Enumeration: com.rti.management.Logger.Category

Method: static Logger.Verbosity getVerbosityByCategory(Logger.Category category)

Method: static void setVerbosityByCategory(Logger.Category category, Logger.Verbosity verbosity)

Sometimes, you're only interested in investigating a specific functional category of the middleware's behavior. Logging at a high level of verbosity across all categories could yield too much output and obscure the information you're looking for.

RTI Message Service recognizes the following logging categories, defined by constants of the **Logger.Category** enumeration. Unlike those of the **Verbosity** enumeration, the **Category** constants are not cumulative; that is, no category includes another.

API	Log messages pertaining to the API layer of *RTI Message Service* (such as method argument validation) are in this category.
COMMUNICATION	Log messages pertaining to data serialization and deserialization and network traffic are in this category.
DATABASE	Log messages pertaining to the internal database in which *RTI Message Service* objects are stored are in this category.
ENTITIES	Log messages pertaining to local and remote JMS objects and to the discovery process are in this category.
PLATFORM	Log messages pertaining to the underlying platform (hardware and OS) on which *RTI Message Service* is running are in this category.

2.1.3 Redirecting Log Output

Method: static java.io.File getOutputFile()

Method: static void setOutputFile(java.io.File out) java.io.throws IOException

By default, *RTI Message Service* logs all output to standard output. If your application launches from a command line terminal and produces no other output, it may be sufficient to simply pipe or redirect all output from the process to a file or another process.

However, if your application has a graphical user interface and hides standard out, of it your application produces many kinds of output and needs to direct them separately, *RTI Message Service* allows you to redirect its output to a specific file.

If the **setOutputFile** method has never been called, **getOutputFile** will return null. Calling **setOutputFile** with a null argument will restore logging to standard output.

2.2 Debugging Connectivity Issues

If you’re observing that data isn’t flowing like you expect from your publishers to your consumers, you may wish to run simple applications to test whether the failure is in your application’s configuration or is related to some deeper problem, like a misconfiguration of your network switching infrastructure. You can use the *rtiddsping* and *rtiddsspy* tools for this purpose. These are simple command-line tools that send and receive messages on your network.

- ❑ The *rtiddsping* tool publishes and subscribes to simple non-JMS "ping" messages to test connectivity between nodes using a variety of QoS settings.
- ❑ The *rtiddsspy* tool subscribes to all JMS messages and displays their contents.

2.2.1 Ping

The *rtiddsping* utility can be run in publisher or subscriber mode to test connectivity between nodes. The packets sent and received by this utility do not contain JMS messages, so the utility will not interfere with the simultaneous use of *RTI Message Service* applications on the network. However, it does support many of the same QoS configurations as *RTI Message Service* itself, allowing you to test more-advanced middleware configurations independently from your application logic.

The *rtiddsping* utility is located in the scripts directory of your *RTI Message Service* installation. It accepts the following configuration options, all of which are optional.

Table 2.1 **Utility Options for rtiddsping**

Option	Description
-help	Prints a help message and exits.
-version	Prints the version and exits.
-Verbosity <NN>	Sets the verbosity level. The range is 0 to 5. • 0 has minimal output and does not echo the fact that data is being sent or received. • 1 prints the most relevant statuses, including the sending and receiving of data. It is the default. • 2 prints a summary of the parameters that are in use and echoes more detailed status messages. • 3-5 mostly affect the verbosity used by the internal *RTI Message Service* modules used to implement rtiddsping. The output is not always readable, its main purpose being to provide information that may be useful to RTI's Support team. Example: `rtiddsping -Verbosity 2`

Table 2.2 **Basic Communication Options for rtiddsping**

Option	Description
-publisher	Causes *rtiddsping* to send ping messages. *This is the default.* Example: `rtiddsping -publisher`
-subscriber	Causes *rtiddsping* to listen for ping messages. This option cannot be specified if **-publisher** is also specified. Example: `rtiddsping -subscriber`
-numSamples <NN>	Sets the number of packets that will be sent by *rtiddsping*. After those samples are sent, *rtiddsping* will exit. If this option is not specified, *rtiddsping* will continue indefinitely. Example: `rtiddsping -numSamples 10`
-sendPeriod <SS>	Sets the period (in seconds) at which *rtiddsping* sends the messages. The default is one second. Example: `rtiddsping -sendPeriod 0.5`
-timeout <SS>	Sets a timeout (in seconds) that will cause *rtiddsping* to exit if no messages are received for a duration that exceeds the timeout. By default, this time is infinite. This option only applies if the **-subscriber** option is also specified. Example: `rtiddsping -timeout 30`

Table 2.3 **QoS Configuration Options for rtiddsping**

Option	Description
-reliable	Configures the reliability QoS for publishing or subscribing. The default setting if this option is not used is best effort. See Chapter 6, "Scalable High-Performance Applications: Message Reliability," in the *User's Manual* for more information about reliability. Example: `rtiddsping -reliable`

Table 2.3 **QoS Configuration Options for rtiddsping**

Option	Description
-durability <KIND>	Sets the durability QoS used for publishing or subscribing. Valid settings for <KIND> are VOLATILE or TRANSIENT_LOCAL (the default). See Chapter 7, "Scalable High-Performance Applications: Durability and Persistence for High Availability," in the *User's Manual* for more information about this QoS. The effect of this setting can only be observed when it is used in conjunction with reliability and a **queueSize** larger than 1. If all these conditions are met, a late-joining subscriber will be able to see up to **queueSize** samples that were previously written by the publisher. Example: `rtiddsping -reliable -durability TRANSIENT_LOCAL`
-queueSize <NN>	Specifies the maximal number of messages to hold in the queue. In the case of the publisher, it affects the messages that are available for a late-joining subscriber. It defaults to 1. See Chapter 6, "Scalable High-Performance Applications: Message Reliability," in the *User's Manual* for more information about queue sizing. Example: `rtiddsping -queueSize 100`
-timeFilter <SS>	Sets the time-based filter QoS for the subscriptions made by *rtiddsping*. This QoS causes *RTI Message Service* to filter out messages that are published at a rate faster than what the filter duration permits. For example if the filter duration is 10 seconds, messages will be printed no faster than once each 10 seconds. See Chapter 3, "Messages and Topics," in the *User's Manual* for more information about time-based filtering. The value 0 indicates no filter and is the default. This option only applies if the **-subscriber** option is also specified. Example: `rtiddsping -subscriber -timeFilter 5.5`

Table 2.3 **QoS Configuration Options for rtiddsping**

Option	Description
-deadline <SS>	This option sets the Deadline QoS for the subscriptions made by *rtiddsping*. It only applies if the **-subscriber** option is also specified. See Chapter 6: Fault Tolerance for more information about this QoS. If this option is not specified, there will be no declared deadline. *Note* that this may cause the subscription QoS to be incompatible with the publisher if the publisher did not specify a **sendPeriod** that is greater than the deadline. If the QoS is incompatible *rtiddsping*, will not receive updates. Each time a missed deadline is detected, *rtiddsping* will print a message that indicates the number of deadlines missed so far. Example: `rtiddsping -subscriber -deadline 3.5`

Table 2.4 **Discovery Options for rtiddsping**

Option	Description
-domainId <domain ID>	Sets the domain ID. The valid range is 0 to 100; the default is 0. For more information about *domains*, see Chapter 4: Data Connectivity. Example: `rtiddsping -domainId 2`
-index <NN>	Sets the connection ID. If it is not -1 (automatic, the default), then it needs to be *different* from the ones used by all other applications in the same computer and domain ID. If this rule is not respected, *rtiddsping* (or the application that starts last) will get an initialization error. For more information about *domains* and the connection ID, see Chapter 4: Data Connectivity. Example: `rtiddsping -domainId 2 -index 1`

Table 2.4 **Discovery Options for rtiddsping**

Option	Description

-peer <PEER>	Specifies a network peer to be used for discovery. Like any *RTI Message Service* application, it defaults to the setting of the environment variable NDDS_DISCOVERY_PEERS or a pre-configured multicast address if the environment is not set.
	The format used for <PEER> is the same used for **initial_peers** and is described in detail in Chapter 4: Data Connectivity. The general format is:
	`NN@TRANSPORT://ADDRESS`
	where:
	❏ ADDRESS is an address (in name form or using the IP notation xxx.xxx.xxx.xxx). ADDRESS may be a multicast address. It cannot be omitted.
	❏ TRANSPORT represents the kind of transport to use.
	❏ NN is the maximum connection ID expected at that location. NN can be omitted and defaults to '4'.
	Valid settings for TRANSPORT are **udpv4** and **shmem**. The default setting if the transport part is omitted is **udpv4**.
	The **-peer** option may be repeated to specify multiple peers.
	Example:
	`rtiddsping -peer 10.10.1.192 -peer mars -peer 4@pluto`

-discoveryTTL <NN>	Sets the TTL (time-to-live) used for multicast discovery. If not specified, *RTI Message Service* will use a default value. The valid range is 0 to 255. The value 0 limits multicast to the node itself (*i.e.* can only discover applications running on the same computer). The setting of '1' limits multicast discovery to computers on the same subnet. Settings greater generally indicate the maximum number of routers that may be traversed (although some routers may be configured differently). Example: `rtiddsping -discoveryTTL 4`
-appId <ID>	Sets the application ID. If unspecified, the system will pick one automatically. *This option is rarely used.*

Table 2.5 **Transport Options for rtiddsping**

Option	Description
-transport <MASK>	A bit-mask that sets the enabled built-in transports. The bit values are: 1 = UDPv4, 2 = shared memory, and 8 = UDPv6. If not specified, the default set of transports are used (UDPv4 + shared memory). Example: `rtiddsping -transport 3`
-multicast <ADDRESS>	Configures ping to receive messages over multicast. The ADDRESS parameter indicates the address to use. ADDRESS must be in the valid range for multicast addresses. For IP version 4, the valid range is 224.0.0.1 to 239.255.255.255. This option only applies if the -subscriber option is also specified. If it is not specified, IP multicast will not be used. Example: `rtiddsping -subscriber -multicast 225.1.1.1`
-msgMaxSize <SIZE>	Configure the maximum packet size allowed by the installed transports. This will be needed if you are using *rtiddsping* to communicate with an application that has set these transport parameters to larger-than-default values.
-shmRcvSize <SIZE>	Increase the size of the shared memory receive buffer. You will need to do this if you are using *rtiddsping* to communicate with an application that has set these transport parameters to larger-than-default values.

2.2.2 Spy

The *rtiddsspy* utility subscribes to JMS messages on any topic and displays their contents so that you can make sure that the messages you *think* you're publishing are actually making it onto the network and to the nodes you expect.

Table 2.6 **Utility Options for rtiddsspy**

Option	Description
-help	Prints a help message and exits.
-hOutput	Print information on the output format used by *rtiddsspy*. This option causes *rtiddsspy* to print to the screen an explanation of the output it produces when it is normally run. After the explanation, is printed it exits. Example: `rtiddsspy -hOutput`
-version	Prints the version and exits.
-Verbosity <NN>	Sets the verbosity level. The range is 0 to 5. ❑ **0** has minimal output and does not echo the fact that data is being sent or received. ❑ **1** prints the most relevant statuses, including the sending and receiving of data. *This is the default.* ❑ **2** prints a summary of the parameters that are in use and echoes more detailed status messages. ❑ **3-5** Mostly affect the verbosity used by the internal *RTI Message Service* modules used to implement *rtiddsping*. The output is not always readable, its main purpose being to provide information that may be useful to RTI's Support team. Example: `rtiddsspy -Verbosity 2`

Table 2.7 **Output Options for rtiddsspy**

Option	Description
-printSample	Prints the contents of each message received.
-showHandle	Causes *rtiddsspy* to print additional information on each message received. The additional information is a hash of the message's key property, which can be used to distinguish among multiple instances published under the same topic name. See Chapter 8, "Scalable High-Performance Applications: Keys," in the *User's Manual* for more information about keys. Example: `rtiddsspy -showHandle`
-topicWidth <WIDTH>	Sets the maximum width of the topic name column. Names wider than this will wrap around, unless -truncate is specified. The value can be in the range [1, 255].
-truncate	Specifies that names exceeding the maximum number of characters should be truncated.

Table 2.8 **QoS Configuration Options for rtiddsspy**

Option	Description
-timeFilter <SS>	Sets the time-based filter QoS for the subscriptions made by *rtiddsspy*. This QoS causes *RTI Message Service* to filter out messages that are published at a rate faster than what the filter duration permits. For example if the filter duration is 10 seconds, messages will be printed no faster than once each 10 seconds. See Chapter 3, "Messages and Topics," in the *User's Manual* for more information about time-based filtering. The value 0 indicates no filter and is the default. Example: `rtiddsspy -timeFilter 5.5`
-deadline <SS>	This option sets the Deadline QoS for the subscriptions made by *rtiddsspy*. See Chapter 6: Fault Tolerance for more information about this QoS. If this option is not specified, there will be no declared deadline. **Note:** this may cause the subscription QoS to be incompatible with the publisher if the publisher does not have a **sendPeriod** that is greater than the deadline. If the QoS is incompatible *rtiddsspy*, will not receive updates. Each time a missed deadline is detected, *rtiddsspy* will print a message that indicates the number of deadlines missed so far. Example: `rtiddsspy -deadline 3.5`

Table 2.9 **Discovery Options for rtiddsspy**

Option	Description
-topicRegex <REGEX>	Subscribe only to topics that match the REGEX regular expression. The syntax of the regular expression is that defined by the POSIX regex function. This option may be repeated to specify topic multiple expressions. If it is not specified, the default value is "*", matching all topic names. *Note* that when typing a regular expression to a command-line shell, some symbols may need to be escaped to avoid interpretation by the shell. In general, it is safest to include the expression in double quotes. Example: `rtiddsspy -topicRegex "Alarm*"`

Table 2.9 **Discovery Options for rtiddsspy**

Option	Description
-domainId <domain ID>	Sets the domain ID. The valid range is 0 to 100; the default is 0. For more information about *domains*, see Chapter 4: Data Connectivity. Example: `rtiddsspy -domainId 2`
-index <NN>	Sets the connection ID. If it is not -1 (automatic, the default), then it needs to be different from the one used by all other applications in the same computer and domain ID. If this rule is not respected, *rtiddsspy* (or the application that starts last) will get an initialization error. For more information about *domains* and the connection ID, see Chapter 4: Data Connectivity. Example: `rtiddsspy -domainId 2 -index 1`

-peer <PEER>	Specifies a network peer to be used for discovery. Like any *RTI Message Service* application, it defaults to the setting of the environment variable NDDS_DISCOVERY_PEERS or a pre-configured multicast address if the environment is not set.
	The format used for **<PEER>** is the same used for **initial_peers** and is described in detail in Chapter 4: Data Connectivity.
	The general format is:
	`NN@TRANSPORT://ADDRESS`
	where:
	❑ ADDRESS is an address (in name form or using the IP notation xxx.xxx.xxx.xxx). ADDRESS may be a multicast address. It cannot be omitted.
	❑ TRANSPORT represents the kind of transport to use
	❑ NN is the maximum connection ID expected at that location. NN can be omitted and is defaulted to '4'.
	Valid settings for TRANSPORT are **udpv4** and **shmem**. The default setting if the transport part is omitted is **udpv4**.
	The **-peer** option may be repeated to specify multiple peers.
	Example:
	`rtiddsspy -peer 10.10.1.192 -peer mars -peer 4@pluto`

Table 2.9 **Discovery Options for rtiddsspy**

Option	Description
-discoveryTTL <NN>	Sets the TTL (time-to-live) used for multicast discovery. If not specified, *RTI Message Service* will use a default value. The valid range is 0 to 255. ❏ 0 limits multicast to the node itself (*i.e.* can only discover applications running on the same computer). ❏ 1 limits multicast discovery to computers on the same subnet. ❏ Values > 1 generally indicate the maximum number of routers that may be traversed (although some routers may be configured differently). Example: `rtiddsspy -discoveryTTL 4`
-appId <ID>	Sets the application ID. If unspecified, the system will pick one automatically. *This option is rarely used.*

Table 2.10 **Transport Options for rtiddsspy**

Option	Description
-transport <MASK>	A bit-mask that sets the enabled built-in transports. The bit values are: 1 = UDPv4, 2 = shared memory, and 8 = UDPv6. If not specified, the default set of transports are used (UDPv4 + shared memory). Example: `rtiddsspy -transport 3`
-msgMaxSize <SIZE>	Configure the maximum packet size allowed by the installed transports. This will be needed if you are using *rtiddsspy* to communicate with an application that has set these transport parameters to larger-than-default values.
-shmRcvSize <SIZE>	Increase the size of the shared memory receive buffer. This will be needed if you are using *rtiddsspy* to communicate with an application that has set these transport parameters to larger-than-default values.

Chapter 3 Network Transport Configuration

This chapter tells you what you need to know in order to configure the way that *RTI Message Service* uses the underlying network—what RTI refers to as the *transport*. RTI supports the following transports out of the box:

- UDP/IP v4
- UDP/IP v6
- Shared memory

RTI Message Service will work automatically with any network adapter on your system whose driver exposes it to the system as an IP interface. This includes not only traditional wired and wireless modems and Ethernet adapters but also higher-performance or more-specialized devices like Infiniband interface cards. It also provides a transport interface into which non-IP transports can be plugged, either by customers or RTI Professional Services. For more information about this facility, please consult your RTI account manager.

This chapter includes the following sections:

- Choosing Your Transports (Section 3.1)
- UDPv4 Configuration (Section 3.2)
- UDPv6 Configuration (Section 3.3)
- Shared Memory Configuration (Section 3.4)

3.1 Choosing Your Transports

By default, *RTI Message Service* will use shared memory to communicate among applications on the same node and UDPv4 to communicate among nodes. However, this configuration may not be appropriate for all applications. For example, you may wish to more-closely simulate the performance of several nodes with a single node by turning of shared memory, or you may wish to use UDPv6 in place of UDPv4. You can also conserve system resources by disabling transports that you know you will never use.

You can activate or deactivate transports on a per-*ConnectionFactory* basis using the Transport Built-in QoS policy. This policy contains a "mask" that specifies the bar ('|')-delimited list of transports to use. The recognized transports are:

- ❏ TRANSPORTBUILTIN_UDPv4
- ❏ TRANSPORTBUILTIN_UDPv6
- ❏ TRANSPORTBUILTIN_SHMEM

Example:

```
<connection_factory name="Example Factory">
    <transport_builtin>
            <mask>
                    TRANSPORTBUILTIN_UDPv4|TRANSPORTBUILTIN_UDPv6
            </mask>
    </transport_builtin>
</connection_factory>
```

Note: The default addresses that the middleware uses for communication rely on the UDPv4 and shared memory transports. If you disable one or both of these transports, you will need to change those addresses to avoid logged warnings and possible communication problems. See the Asymmetric Discovery Configuration (Section 4.2.3) for more information.

You can also configure transports on a per-*MessageProducer* or per-*MessageConsumer* basis using the Transport Selection QoS policy. This policy contains a list of strings indicating the transports to be used.

Example:

```
<topic name”Example Topic”>
    <consumer_defaults>
            <transport_selection>
                    <enabled_transports>
                            <element>builtin.udpv4</element>
                            <element>builtin.udpv6</element>
                            <element>builtin.shmem</element>
                    </enabled_transports>
            </transport_selection>
    </consumer_defaults>
</topic>
```

Because an empty list (i.e., no **<element>** elements is not useful, it is used as a sentinel: it indicates that all transports that are active on the containing *Connection* will be used by the producer or consumer.

The UDP/IP v4 transport supports unicast and multicast communication. *RTI Message Service* uses a reliable protocol called Real-Time Publish Subscribe (RTPS) on top of UDP to provide reliability and other services not available in UDP itself.

3.2 UDPv4 Configuration

This transport plug-in uses UDPv4 sockets to send and receive messages. It supports both unicast and multicast communications. By default, it will use all interfaces that it finds enabled and “UP” at *Connection* instantiation time to send and receive messages.

RTI Message Service implicitly initializes this plug-in if it is specified in Transport Built-in QoS policy described above. You can configure it to only use unicast or only use multicast, see the **unicast_enabled** and **multicast_enabled** properties described below.

In addition, you can configure this plug-in to selectively use the network interfaces of a node (and restrict a plug-in from sending multicast messages on specific interfaces) by specifying the "white" and "black" lists (the **allow_interfaces**, **deny_interfaces**, **allow_multicast_interfaces**, and **deny_multicast_interfaces** properties).

Configure the UDPv4 transport using the Property QoS policy of a *ConnectionFactory* like this:

```
<connection_factory name="Example Factory">
    <property>
           <value>
                  <element>
                         <name>name1</name>
                         <value>value1</value>
                  </element>
                  <element>
                         <name>name2</name>
                         <value>value2</value>
                  </element>
           </value>
    </property>
</connection_factory>
```

Table 3.1 on page 3-5 lists the UDPv4 connection properties.

Each connection will open up to four UDP/IP ports:

❑ The *meta-traffic unicast port* is used to exchange discovery-related meta-traffic using unicast. This port will not be used if unicast traffic has been disabled.

❑ The *meta-traffic multicast port* is used to exchange discovery meta-traffic using multicast. This port will not be used if multicast traffic has been disabled.

❑ The *user traffic unicast port* is used to exchange application data using unicast. This port will not be used if unicast traffic has been disabled.

❑ The *user traffic multicast port* is used to exchange application data using multicast. This port will not be used if multicast traffic has been disabled.

The numbers of these ports are described in Segregate Systems and Subsystems into Domains (Section 4.1)

3.3 UDPv6 Configuration

This transport plug-in uses UDPv6 sockets to send and receive messages. It supports both unicast and multicast communications. By default, it will use all interfaces that it finds enabled and "UP" at *Connection* instantiation time to send and receive messages.

Table 3.1 **UDPv4 Connection Properties**

Property Name	Description
dds.transport.UDPv4.builtin. parent. message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plug-in.
dds.transport.UDPv4.builtin. parent. allow_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, allow the use of only these interfaces; otherwise allow the use of all interfaces. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"
dds.transport.UDPv4.builtin. parent.deny_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, deny the use of these interfaces. This "black" list is applied after the **allow_interfaces** and filters out the interfaces that should not be used. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"
dds.transport.UDPv4.builtin. parent. allow_multicast_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, allow the use of multicast only these interfaces; otherwise allow the use of all the allowed interfaces. This "while" list sub-selects from the allowed interfaces obtained after applying the **allow_interfaces** "white" list and the **deny_interfaces** "black" list. If this list is empty, all the allowed interfaces will be potentially used for multicast. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"

Table 3.1 **UDPv4 Connection Properties**

Property Name	Description
dds.transport.UDPv4.builtin. parent.deny_multicast_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, deny the use of those interfaces for multicast. This "black" list is applied after the **allow_multicast_interfaces** and filters out the interfaces that should not be used for multicast. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"
dds.transport.UDPv4.builtin. send_socket_buffer_size	Size in bytes of the send buffer of a socket used for sending. On most operating systems, **setsockopt()** will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to **message_size_max**. If you configure this parameter to be equal to the OS default, then **setsockopt()** (or equivalent) will not be called to size the send buffer of the socket.
dds.transport.UDPv4.builtin. recv_socket_buffer_size	Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, setsockopt() will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to **message_size_max**. If it is set to the OS default, then **setsockopt()** (or equivalent) will not be called to size the receive buffer of the socket.
dds.transport.UDPv4.builtin. unicast_enabled	Allows the transport plug-in to use unicast for sending and receiving. The user can turn on or off the use of unicast UDP for this plug-in. By default, it will be turned on. Also by default, it will use all the allowed network interfaces that it finds up and running when the plug-in is instanced.

Table 3.1 **UDPv4 Connection Properties**

Property Name	Description
dds.transport.UDPv4.builtin. multicast_enabled	Allows the transport plug-in to use multicast for sending and receiving. The user can turn on or off the use of multicast UDP for this plug-in. By default it will be turned on. Also by default, it will use the all network interfaces allowed for multicast that it finds up and running when the plug-in is instanced.
dds.transport.UDPv4.builtin. multicast_ttl	Value for the time-to-live parameter for all multicast sends. This is used to set the TTL of multicast packets sent by this transport plug-in.
dds.transport.UDPv4.builtin. multicast_loopback_disabled	Prevents the transport plug-in from putting multicast packets onto the loopback interface. This will prevent other applications on the same node (including itself) from receiving those packets. This is set to 0 by default, so multicast loopback is enabled. Turning off multicast loopback (set to 1) may result in minor performance gains when using multicast.

Table 3.1 **UDPv4 Connection Properties**

Property Name	Description
dds.transport.UDPv4.builtin. ignore_loopback_interface	Prevents the transport plug-in from using the IP loopback interface. Three values are allowed: ❑ 0: Enable local traffic via this plug-in. This plug-in will only use and report the IP loopback interface only if there are no other network interfaces (NICs) up on the system. ❑ 1: Disable local traffic via this plug-in. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plug-in like Shared Memory to talk instead of the IP loopback. ❑ -1: Automatic. Let *RTI Message Service* decide among the above two choices. If a shared memory transport plug-in is available for local traffic, the effective value is 1 (i.e., disable UPv4 local traffic). Otherwise, the effective value is 0, i.e., use UDPv4 for local traffic also.

Table 3.1 **UDPv4 Connection Properties**

Property Name	Description
dds.transport.UDPv4.builtin. ignore_nonrunning_interfaces	Prevents the transport plug-in from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed: 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. This is the default. 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.

RTI Message Service implicitly initializes this plug-in if it is specified in the Transport Built-in QoS policy. *This is not the default configuration.*

You can configure this plug-in to only use unicast or only use multicast; see the **unicast_enabled** and **multicast_enabled** properties described below.

In addition, you can configure this plug-in to selectively use the network interfaces of a node (and restrict it from sending multicast messages on specific interfaces) by specifying the "white" and "black" lists (the **allow_interfaces**, **deny_interfaces**, **allow_multicast_interfaces**, and **deny_multicast_interfaces** properties).

Configure the UDPv6 transport using the Property QoS policy of a *ConnectionFactory* like this:

```
<connection_factory name="Example Factory">
    <property>
            <value>
                    <element>
                            <name>name1</name>
                            <value>value1</value>
                    </element>
                    <element>
                            <name>name2</name>
                            <value>value2</value>
                    </element>
            </value>
    </property>
</connection_factory>
```

Table 3.2 on page 3-11 lists the UDPv6 Connection Properties.

Each connection will open up to four UDP/IP ports:

- ❏ The *meta-traffic unicast port* is used to exchange discovery-related meta-traffic using unicast. This port will not be used if unicast traffic has been disabled.
- ❏ The *meta-traffic multicast port* is used to exchange discovery meta-traffic using multicast. This port will not be used if multicast traffic has been disabled.
- ❏ The *user traffic unicast port* is used to exchange application data using unicast. This port will not be used if unicast traffic has been disabled.
- ❏ The *user traffic multicast port* is used to exchange application data using multicast. This port will not be used if multicast traffic has been disabled.

The numbers of these ports are described in Section 4.1, "Segregate Systems and Subsystems into Domains," in the User's Manual.

Table 3.2 **UDPv6 Connection Properties**

Property Name	Description
dds.transport.UDPv6.builtin. parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plug-in.
dds.transport.UDPv6.builtin. parent.allow_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, allow the use of multicast only these interfaces; otherwise allow the use of all the allowed interfaces. This "while" list sub-selects from the allowed interfaces obtained after applying the **allow_interfaces** "white" list and the **deny_interfaces** "black" list. If this list is empty, all the allowed interfaces will be potentially used for multicast. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"
dds.transport.UDPv6.builtin. parent.deny_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, deny the use of these interfaces. This "black" list is applied after the **allow_interfaces** and filters out the interfaces that should not be used. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"
dds.transport.UDPv6.builtin. parent. allow_multicast_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, allow the use of multicast only these interfaces; otherwise allow the use of all the allowed interfaces. This "while" list sub-selects from the allowed interfaces obtained after applying the **allow_interfaces** "white" list and the **deny_interfaces** "black" list. If this list is empty, all the allowed interfaces will be potentially used for multicast. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"

Table 3.2 **UDPv6 Connection Properties**

Property Name	Description
dds.transport.UDPv6.builtin. parent. deny_multicast_interfaces	A list of strings, each identifying a range of interface addresses. If the list is non-empty, deny the use of those interfaces for multicast. This "black" list is applied after the **allow_multicast_interfaces** and filters out the interfaces that should not be used for multicast. Interfaces should be specified as comma-separated strings, with each comma delimiting an interface. For example, "127.0.0.1,eth0"
dds.transport.UDPv6.builtin. send_socket_buffer_size	Size in bytes of the send buffer of a socket used for sending. On most operating systems, **setsockopt()** will be called to set the SENDBUF to the value of this parameter. This value must be greater than or equal to **message_size_max**. The maximum value is operating system-dependent. If you configure this parameter to be the OS default, then **setsockopt()** (or equivalent) will not be called to size the send buffer of the socket.
dds.transport.UDPv6.builtin. recv_socket_buffer_size	Size in bytes of the receive buffer of a socket used for receiving. On most operating systems, **setsockopt()** will be called to set the RECVBUF to the value of this parameter. This value must be greater than or equal to **message_size_max**. The maximum value is operating system-dependent. If it is set to the OS default, then **setsockopt()** (or equivalent) will not be called to size the receive buffer of the socket.
dds.transport.UDPv6.builtin. unicast_enabled	Allows the transport plug-in to use unicast for sending and receiving. By default, unicast will be turned on. Also by default, the transport will use all the allowed network interfaces that it finds up and running when the Connection is instanced.
dds.transport.UDPv6.builtin. multicast_enabled	Allows the transport plug-in to use multicast for sending and receiving. By default, multicast will be turned on. Also by default, the transport will use all network interfaces allowed for multicast that it finds up and running when the plug-in is instanced.

Table 3.2 **UDPv6 Connection Properties**

Property Name	Description
dds.transport.UDPv6.builtin.multicast_ttl	Value for the time-to-live parameter for all multicast sends using this plug-in.
dds.transport.UDPv6.builtin.multicast_loopback_disabled	Prevents the transport plug-in from putting multicast packets onto the loopback interface. This will prevent other applications on the same node (including itself) from receiving those packets. This is set to 0 by default, so multicast loopback is enabled. Turning off multicast loopback (set to 1) may result in minor performance gains when using multicast.
dds.transport.UDPv6.builtin.ignore_loopback_interface	Prevents the transport plug-in from using the IP loopback interface. Three values are allowed: ❏ 0: Enable local traffic via this plug-in. This plug-in will only use and report the IP loopback interface only if there are no other network interfaces (NICs) up on the system. ❏ 1: Disable local traffic via this plug-in. Do not use the IP loopback interface even if no NICs are discovered. This is useful when you want applications running on the same node to use a more efficient plug-in like Shared Memory to talk instead of the IP loopback. ❏ -1: Automatic. Let *RTI Message Service* decide among the above two choices. If a shared memory transport plug-in is available for local traffic, the effective value is 1 (i.e., disable UPv4 local traffic). Otherwise, the effective value is 0, i.e., use UDPv4 for local traffic also.

Table 3.2 **UDPv6 Connection Properties**

Property Name	Description
dds.transport.UDPv6.builtin. ignore_nonrunning_interfaces	Prevents the transport plug-in from using a network interface that is not reported as RUNNING by the operating system. The transport checks the flags reported by the operating system for each network interface upon initialization. An interface which is not reported as UP will not be used. This property allows the same check to be extended to the IFF_RUNNING flag implemented by some operating systems. The RUNNING flag is defined to mean that "all resources are allocated", and may be off if there is no link detected, e.g., the network cable is unplugged. Two values are allowed: 0: Do not check the RUNNING flag when enumerating interfaces, just make sure the interface is UP. This is the default. 1: Check the flag when enumerating interfaces, and ignore those that are not reported as RUNNING. This can be used on some operating systems to cause the transport to ignore interfaces that are enabled but not connected to the network.
dds.transport.UDPv6.builtin. enable_v4mapped	Specify whether the UDPv6 transport will process IPv4 addresses. Set this to 1 to turn on processing of IPv4 addresses. Note that this may make the UDPv6 transport incompatible the UDPv4 transport within the same Connection.

3.4 Shared Memory Configuration

This plug-in uses system shared memory to send messages between processes on the same node.

Note: For the sake of efficiency, activating the shared memory transport will deactivate the use of IP loopback communication. This optimization prevents the middleware from sending duplicate copies of each message bound for the local host over both transports, only to discard one copy of each message upon reception. *It is therefore critical* for all applications on the same host that you intend to communicate together to have the same shared memory and UDP transport configurations.

3.4.1 Compatibility of Sender and Receiver Transports

Opening a receiver "port" on shared memory corresponds to creating a shared memory segment using a name based on the port number. The transport plug-in's properties are embedded in the shared memory segment.

When a sender tries to send to the shared memory port, it verifies that properties of the receiver's shared memory transport are compatible with those specified in its transport plug-in. If not, the sender will fail to attach to the port and will output messages such as below (with numbers appropriate to the properties of the transport plug-ins involved).

```
NDDS_Transport_Shmem_attachShmem:failed to initialize incompatible
properties
NDDS_Transport_Shmem_attachShmem:countMax 0 > -19417345 or max size
-19416188 > 2147482624
```

In this scenario, the properties of the sender or receiver transport plug-in instances should be adjusted so that they are compatible.

3.4.2 Crashing and Restarting Programs

If a process using shared memory crashes (say because the user typed in ^C), resources associated with its shared memory ports may not be properly cleaned up. Later, if another *RTI Message Service* process needs to open the same ports (say, the crashed program is restarted), it will attempt to reuse the shared memory segment left behind by the crashed process.

The reuse is allowed if and only if the properties of the transport plug-in are compatible with those embedded in the shared memory segment (*i.e.*, of the original creator). Otherwise, the process will fail to open the ports and will output messages such as below (with numbers appropriate to the properties of the transport plug-ins involved).

```
NDDS_Transport_Shmem_create_recvresource_rrEA:failed to initialize
shared memory resource Cannot recycle existing shmem: size not com-
patible for key 0x1234
```

In this scenario, the shared memory segments must be cleaned up using appropriate platform specific commands. For details, please refer to the Release Notes.

3.4.3 Shared Resource Keys

The transport uses the shared memory segment keys given by this formula:

```
0x400000 + port
```

The transport also uses signaling shared semaphore keys given by this formula:

```
0x800000 + port
```

The transport also uses mutex shared semaphore keys given by this formula:

```
0xb00000 + port
```

where the port is a function of the domain ID and the connection ID, as described in Chapter 4: Data Connectivity.

3.4.4 Configuration

Configure the shared memory transport using the Property QoS policy of a *Connection-Factory* like this:

```
<connection_factory name="Example Factory">
    <property>
           <value>
                  <element>
                         <name>name1</name>
                         <value>value1</value>
                  </element>
                  <element>
                         <name>name2</name>
                         <value>value2</value>
                  </element>
           </value>
    </property>
</connection_factory>
```

Table 3.3 lists the shared memory configuration properties.

Table 3.3 **Shared Memory Configuration Properties**

Property Name	Description
dds.transport.shmem.builtin. parent.message_size_max	The maximum size of a message in bytes that can be sent or received by the transport plug-in.
dds.transport.shmem.builtin. received_message_count_max	Number of messages that can be buffered in the receive queue. This does not guarantee that the transport plug-in will actually be able to buffer **received_message_count_max** messages of the maximum size set in **message_size_max**. The total number of bytes that can be buffered is actually controlled by **receive_buffer_size**.

Table 3.3 **Shared Memory Configuration Properties**

Property Name	Description

dds.transport.shmem.builtin.
receive_buffer_size
	The total number of bytes that can be buffered in the receive queue.
	
	This number controls how much memory is allocated by the plug-in for the receive queue. The actual number of bytes allocated is:
	
	`size =`
	`receive_buffer_size + message_size_max +`
	`received_message_count_max * fixedOverhead`
	
	where **fixedOverhead** is some small number of bytes used by the queue data structure. The following rules are noted:
	
	If **receive_buffer_size** < (**message_size_max** * **received_message_count_max**), then the transport plug-in will not be able to store received_message_count_max messages of size receive_buffer_size.
	
	If **receive_buffer_size** > (**message_size_max** * **received_message_count_max**), then there will be memory allocated that cannot be used by the plug-in and thus wasted.
	
	To optimize memory usage, you are allowed to specify a size for the receive queue less than that required to hold the maximum number of messages which are all of the maximum size.
	
	In most situations, the average message size may be far less than the maximum message size. So for example, if the maximum message size is 64 K bytes, and the user configures the plug-in to buffer at least 10 messages, then 640 K bytes of memory would be needed if all messages were 64 K bytes. Should this be desired, then **receive_buffer_size** should be set to 640 K bytes.
	
	However, if the average message size is only 10 K bytes, then the user could set the **receive_buffer_size** to 100 K bytes. This allows the user to optimize the memory usage of the plug-in for the average case and yet allow the plug-in to handle the extreme case.
	
	NOTE, the queue will always be able to hold 1 message of **message_size_max** bytes, no matter what the value of **receive_buffer_size** is.

Chapter 4 Data Connectivity

Sometimes application designers and system administrators require greater control over which messaging objects will communicate than can be controlled simply by topic names.

- ❏ Multiple distinct applications may exist independently on the same physical network with the requirement that they remain entirely isolated from one another.
- ❏ You may require control over the network addresses used by the middleware or over the style of addressing used (*e.g.* unicast or multicast).
- ❏ Messages pertaining to different data objects (e.g., stock symbols or radar tracks) may be destined for different multicast addresses and different consumers in order to control the load on your network.
- ❏ You may wish to restrict the flow of discovery-related traffic in order to decrease the demands on your network and speed up your system's start-up time.

This chapter describes how to achieve these and similar use cases. It includes the following sections:

4.1 Segregate Systems and Subsystems into Domains

Every *Connection* in an *RTI Message Service* application belongs to exactly one *domain*. A domain is a virtual network that overlays your physical network; connections belonging to different domains will never exchange messages or even know of the existence of one another. Domains therefore constitute the coarsest-granularity mechanism for isolating subsystems—or entire distributed applications—from one another.

All connections created from the same factory will belong to the same domain. The domain is identified by a non-negative integer, its *domain ID*, that is specified in your configuration file:

```
<connection_factory>
    <domain_id>2</domain_id>
</connection_factory>
```

If no domain ID is specified, it takes the default value zero (0).

A single application program may create connections in any number of domains. It may do so, for example, to bridge selected messages between different domains.

It is also possible for an application program to create multiple connections in the same domain. However, RTI recommends that applications do this only when there is a well-defined reason to do so. Connections are heavyweight objects that internally allocate threads, sockets, and other system resources; creating a large number of them can adversely impact performance and resource usage.

Every *Connection* object is internally identified by a *connection ID* that is unique within a given host. (This additional level of disambiguation is necessary when an application chooses to create multiple connections within the same domain.) The middleware will assign this ID on your behalf; in most cases, you do not need to set it or even know what its value is. However, it is used to determine which UDP ports the connection will attempt to open, so if you need to determine the connection's port usage deterministically, you will need to set it manually.

RTI Message Service uses the ports described in Table 4.1. Table 4.2 describes the parameters used in Table 4.1.

Table 4.1 **Ports Used by RTI Message Service**

Port	Formula
Meta-traffic unicast port—used to exchange discovery meta-traffic using unicast.	port_base + (domain_id_gain * domain_id) + (connection_id_gain * connection_id) + builtin_unicast_port_offset = metatraffic_unicast_port
Meta-traffic multicast port—used to exchange discovery meta-traffic using multicast.	port_base + (domain_id_gain * domain_id) + builtin_multicast_port_offset = metatraffic_multicast_port
User traffic unicast port—used to exchange application data using unicast.	port_base + (domain_id_gain * domain_id) + (connection_id_gain * connection_id) + user_unicast_port_offset = usertraffic_unicast_port
User traffic multicast port—used to exchange application data using multicast.	port_base + (domain_id_gain * domain_id) + user_multicast_port_offset = usertraffic_multicast_port

Table 4.2 **Port Calculation Parameters**

Parameter	Description
port_base	All mapped well-known ports are offset by this value. **[default]** 7400 **[range]** >= 1, but resulting ports must be within the range imposed by the underlying transport.
domain_id_gain	Multiplier of the domain_id. Together with connection_id_gain, it determines the highest domain ID and connection ID allowed on this network. See Section 4.1.1 for details.

Table 4.2 **Port Calculation Parameters**

Parameter	Description
connection_id_ gain	Multiplier of the connection_id. See domain_id_gain for its implications on the highest domain ID and connection ID allowed on this network. Additionally, connection_id_gain also determines the range of builtin_unicast_port_offset and user_unicast_port_offset. connection_id_gain > abs(builtin_unicast_port_offset - user_unicast_port_offset) [default] 2 [range] > 0, but resulting ports must be within the range imposed by the underlying transport. See Section 4.1.1 for details.
builtin_multicast_ port_offset	Additional offset for meta-traffic multicast port. It must be unique from other port-specific offsets. [default] 0 [range] >= 0, but resulting ports must be within the range imposed by the underlying transport.
builtin_unicast_ port_offset	Additional offset for meta-traffic unicast port. It must be unique from other port-specific offsets. [default] 10 [range] >= 0, but resulting ports must be within the range imposed by the underlying transport.
user_multicast_ port_offset	Additional offset for user traffic multicast port. It must be unique from other port-specific offsets. [default] 1 [range] >= 0, but resulting ports must be within the range imposed by the underlying transport.
user_unicast_ port_offset	Additional offset for user traffic unicast port. It must be unique from other port-specific offsets. [default] 11 [range] >= 0, but resulting ports must be within the range imposed by the underlying transport.

To summarize:

- ❑ Set the domain ID if you need to isolate multiple systems and/or subsystems from one another, even when they exist simultaneously on the same physical network.
- ❑ Set the connection ID if you need to be able to predict the network ports that will be used by a given connection.
- ❑ Set one or more of the fields of the **rtps_well_known_ports** element if you need to select those ports specifically—for example, if you need to route *RTI Message Service* traffic through a firewall, and the range of ports allowed through are not under your control, or if another application requires all of the ports in the default range.

Example:

```
<connection_factory>
    <domain_id>1</domain_id>
    <wire_protocol>
            <connection_id>1</connection_id>
            <rtps_well_known_ports>
                    <port_base>7400</port_base>
                    <domain_id_gain>250</domain_id_gain>
                    <connection_id_gain>2</connection_id_gain>
                    <builtin_multicast_port_offset>
                           0
                    </builtin_multicast_port_offset>
                    <builtin_unicast_port_offset>
                           10
                    </builtin_unicast_port_offset>
                    <user_multicast_port_offset>
                           1
                    </user_multicast_port_offset>
                    <user_unicast_port_offset>
                           11
                    </user_unicast_port_offset>
            </rtps_well_known_ports>
    </wire_protocol>
</connection_factory>
```

4.1.1 domain_id_gain and connection_id_gain

In general, there are two ways to set up **domain_id_gain** and **connection_id_gain** parameters.

If **domain_id_gain** > **connection_id_gain**, it results in a port mapping layout where all connections within a single domain occupy a consecutive range of **domain_id_gain** ports. Precisely, all ports occupied by the domain fall within:

(**port_base** + (**domain_id_gain** * **domain_id**))

and

(**port_base** + (**domain_id_gain** * (**domain_id** + 1)) - 1)

In such a case, the highest domain ID is limited only by the underlying transport's maximum port. The highest connection ID, however, must satisfy:

max_connection_id < (**domain_id_gain** / **connection_id_gain**)

However, if **domain_id_gain** <= **connection_id_gain**, it results in a port mapping layout where a given domain's connections occupy ports spanned across the entire valid port range allowed by the underlying transport. For instance, it results in the following potential mapping:

Mapped Port	Domain ID	Connection ID
higher port number	1	2
	0	2
	1	1
	0	1
	1	0
lower port number	0	0

In this case, the highest connection_id is limited only by the underlying transport's maximum port. The highest domain_id, however, must satisfy:

max_domain_id < (connection_id_gain / domain_id_gain)

Additionally, domain_id_gain also determines the range of the port-specific offsets.

domain_id_gain >
abs(builtin_multicast_port_offset - user_multicast_port_offset)

domain_id_gain >
abs(builtin_unicast_port_offset - user_unicast_port_offset)

Violating this may result in port aliasing and undefined discovery behavior.

[default] 250

[range] > 0, but resulting ports must be within the range imposed by the underlying transport

4.2 Tune Discovery for Faster Startup and Improved Scalability

RTI Message Service automatically maintains information about the connections, producers, and consumers on the network with a dynamic process called *discovery*. This section describes how to configure the middleware's built-in discovery service; before you continue, be sure you have read Section 2.4, Introduction to Peer-to-Peer Discovery, in the User's Manual.

By default, the discovery process uses IP multicast. If your application exists on a single subnet, you probably don't need to do anything. However, you may need to change your discovery configuration if any of the following conditions apply to you:

- ❑ You need to change the addresses used by the discovery service. You want either to change the multicast address or to use unicast addresses instead.
- ❑ The middleware's default balance between responsiveness and network bandwidth utilization does not meet your needs. You either need to make the middleware respond to topology changes more quickly (possibly at the cost of making the product more "chatty") or to decrease the amount of bandwidth used by the discovery process (possibly at the cost of decreasing responsiveness).
- ❑ You want to custom-tune which applications discover which other applications in order to optimize network bandwidth utilization and thereby increase performance in a large system. Doing so will require careful system-wide address selection and may require asymmetric configuration to obtain the desired behavior.

4.2.1 Introduction: Discovery Announcements

The discovery process occurs at the granularity of a single *Connection* and the message producers and consumers created from it. The remote connections known to a given local connection are referred to as its *peers*. The discovery process between a connection and its peers can be thought of as two simultaneous sub-processes:

- **Connection Discovery Protocol**: Because the discovery process is dynamic—that is, applications can join and leave the network at any time—connections announce *themselves* to one another in a best-effort fashion. They cannot know which potential peers exist in actuality, and so they rely on a kind of "probabilistic reliability," sending multiple announcements to one another to make it very likely that at least one announcement will go through before an application-specified timeout expires.
- **Endpoint Discovery Protocol**: Once a peer *Connection* has been discovered, the local *Connection* will open reliable data channels to communicate information about message producers and consumers. As it receives each remote endpoint declaration, it will look for matches among its own endpoints: producers and consumers on the same topic with compatible configurations.

Communication will not occur between a given pair of peers unless and until each has discovered the other and matched the relevant endpoints.

4.2.1.1 Connection Discovery

Connections send announcements to one another under well-defined circumstances. These announcements are sent to all known peer connections.

When it is first created…

A *Connection* sends a certain number of "initial" announcements when it is first created. These declarations are separated by a random amount of time in between application-specified minimum and maximum values. These parameters are part of the DiscoveryConfig QoS policy; the following XML shows the default values:

```
<connection_factory name="Example Factory">
    <discovery_config>
            <initial_connection_announcements>
                    5
            </initial_connection_announcements>
            <min_initial_connection_announcement_period>
                    <sec>1</sec>
                    <nanosec>0</nanosec>
            </min_initial_connection_announcement_period>
            <max_initial_connection_announcement_period>
                   <sec>1</sec>
                   <nanosec>0</nanosec>
            </max_initial_connection_announcement_period>
    </discovery_config>
</connection_factory>
```

Upon receiving an announcement from a never-before-seen peer *Connection*…

When a *Connection* receives an announcement from a peer connection from which it has never before received an announcement, it will resend its own announcement to encourage reciprocal discovery to complete more quickly.

Periodically…

Each *Connection* resends its own announcement at a rate specified in the DiscoveryConfig QoS policy. The following XML shows the default value:

```
<connection_factory name="Example Factory">
    <discovery_config>
            <connection_liveliness_assert_period>
                   <sec>30</sec>
                   <nanosec>0</nanosec>
            </connection_liveliness_assert_period>
    </discovery_config>
</connection_factory>
```

When it is disposed…

When the *Connection* is finally garbage collected, it will send out a final announcement.

If a *Connection* receives no announcements from one of its peers for an application-configured period of time, it will consider that peer to have left the network, and will purge all information about it and its producers and consumers from its internal database. Communication with that peer will cease at that time. This timeout is part of the DiscoveryConfig QoS policy; the following XML shows the default value:

```
<connection_factory name="Example Factory">
    <discovery_config>
            <connection_liveliness_lease_duration>
                   <sec>100</sec>
                   <nanosec>0</nanosec>
            </connection_liveliness_lease_duration>
    </discovery_config>
</connection_factory>
```

4.2.1.2 Endpoint Discovery

As soon as a *Connection* has discovered a new peer *Connection*, it will send to its new peer data about its own endpoints—message producers and consumers. That allows endpoint discovery to be carried out in parallel with *Connection* discovery; it need not wait for *Connection* discovery to complete.

Unlike during the *Connection* discovery process, endpoint discovery is conducted between *known* peers, not just *potential* peers. That means that the middleware can use true reliability, not just the probabilistic model used for *Connection* discovery. That means that producer and consumer announcements will be sent only when endpoints are created and deleted; periodic announcements are not necessary.

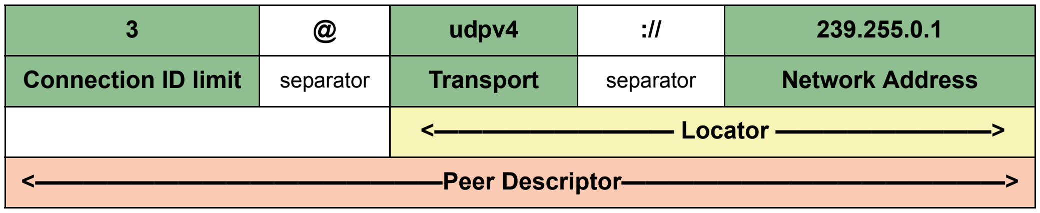
When a remote *MessageProducer*/*MessageConsumer* is discovered, *the local Connection* determines if it has a matching *MessageConsumer*/*MessageProducer*. A 'match' between the local and remote entities occurs only if the *MessageConsumer* and *MessageProducer* have the same *Topic*, same key setting, and compatible QoS policies.

4.2.2 Addresses Used for Discovery

There are two categories of addresses to consider: (*1*) those at which a *Connection* expects to find its remote peers and (*2*) those at which those peers can contact that local *Connection*. The former are referred to as the *Connection*'s "initial peers"; they represent potential communication partners. The latter are referred to a *Connection*'s "receive locators": the network addresses and ports at which it is listening for incoming data.

4.2.2.1 Peer Descriptor Format

A locator—the combination of network transport and address that represents a physical destination for network packets—can be represented in a string form. A "peer descriptor" string represents a range of potential connections that could exist at a given locator. This string format is used to describe both initial peer locators and receive locators.



A peer descriptor consists of two components:

- *[optional]* A connection ID limit, which specifies the maximum connection ID that will be contacted by the *RTI Message Service* discovery process at the given locator. *For unicast locators*: If it is omitted, a default value of 4 is implied. *For multicast locators*: It is ignored, and therefore should be omitted from multicast peer descriptors.
- A locator, as described in Locator Format (Section 4.2.2.1.1).

These are separated by the '@' character. The separator should be omitted if a connection ID limit is not explicitly specified.

4.2.2.1.1 Locator Format

A locator specifies a transport and an address. These are combined in a string form that resembles a URL.

A locator consists of:

- *[optional]* Transport name. This identifies the transport plug-in that will be used to parse the address portion of the locator.
- *[optional]* An address specified in IPv4 or IPv6 format.

These are separated by the "**://**" string. The separator is specified if and only if a transport name is specified.

If a transport name is specified, the address may be omitted; in that case, all the unicast addresses associated with the transport are implied. Thus, a locator string may specify several addresses.

If an address is specified, the transport name and the separator string may be omitted; in that case, all the available transport plug-ins may be used to parse the address string.

The transport names for the built-in transport plug-ins are:

- **shmem** - Shared Memory Transport
- **udpv4** - UDP/IP v4 Transport
- **udpv6** - UDP/IP v6 Transport

4.2.2.1.1 Peer Descriptor Examples

Table 4.3 **NDDS_DISCOVERY_PEERS Environment Variable Examples**

Peer Descriptor	Description of Host(s)
239.255.0.1	multicast
localhost	localhost
192.168.1.1	10.10.30.232 (IPv4)
FAA0::1	FAA0::0 (IPv6)
himalaya,gangotri	himalaya and gangotri
1@himalaya,1@gangotri	himalaya and gangotri (with a maximum connection ID of 1 on each host)
FAA0::0localhost	FAA0::0localhost (could be a UDPv4 transport plug-in registered at network address of FAA0::0) (IPv6)
udpv4://himalaya	himalaya accessed using the "udpv4" transport plug-ins) (IPv4)
udpv4://FAA0::0localhost	localhost using the "udpv4" transport plug-ins) registered at network address FAA0::0
udpv4://	all unicast addresses accessed via the "udpv4" (UDPv4) transport plug-ins)
shmem://	all unicast addresses accessed via the "shmem" (shared memory) transport plug-ins
shmem://FCC0::0	all unicast addresses accessed via the "shmem" (shared memory) transport plug-ins registered at network address FCC0::0

4.2.2.2 Initial Peers

A *Connection*'s "initial peers" list is a collection of strings in peer descriptor format (see section Peer Descriptor Format (Section 4.2.2.1)) that describe peer *Connection*'s that *may* exist on the network. A *Connection* will attempt to carry out the Connection Discovery Protocol with all of those potential remote *Connection*'s. When a remote *Connection* is actually discovered—that is, a reciprocal *Connection* announcement is received—it will begin carrying out the Endpoint Discovery Protocol with that remote *Connection*.

The initial peers list is part of the Discovery QoS policy. The following shows the default value as it would be described in a configuration file:

```
<connection_factory name="Example Factory">
    <discovery>
           <initial_peers>
                  <!-- UDPv4 multicast address: -->
                  <element>udpv4://239.255.0.1</element>
                  <!-- UDPv4 localhost unicast address: -->
                  <element>4@udpv4://127.0.0.1</element>
                  <!-- Shared memory: -->
                  <element>shmem://</element>
           </initial_peers>
    </discovery>
</connection_factory>
```

4.2.2.3 Receive Locators

Connection declarations contain, among other things, a list of the locators at which the *Connection* can be contacted. (See Locator Format (Section 4.2.2.1.1) for more information.) This list will contain:

- ❏ If the UDP/IP v4 transport is active, the UDPv4 unicast addresses of all network interfaces configured for use with *RTI Message Service*.
- ❏ If the UDP/IP v6 transport is active, the UDPv6 unicast addresses of all network interfaces configured for use with *RTI Message Service*.
- ❏ If the shared memory transport is active, the shared memory segment corresponding to the Connection's domain. For more information about domains, see Segregate Systems and Subsystems into Domains (Section 4.1) .

If the UDPv4 and/or UDPv6 transports are active, an optional IP multicast address. This address, called the multicast receive address, is configured with the Discovery QoS policy; the following XML shows the default value:

```
<connection_factory name="Example Factory">
    <discovery >
           <multicast_receive_addresses>
                  <element>udpv4://239.255.0.1</element>
           </multicast_receive_addresses>
    </discovery >
</connection_factory>
```

Note that the multicast receive address is specified in the configuration file as if it were a list of addresses. However, only a single address is currently supported; any subsequent list items will be ignored.

For more information about network transport configuration, see Chapter 3: Network Transport Configuration.

4.2.2.4 Addressing and Transports

Your choice of initial peers and multicast receive address are not decoupled from your choice of network transports. See Chapter 3: Network Transport Configuration.

- ❑ If you disable the UDPv4 or shared memory transport but leave your initial peer list unchanged, the middleware will log warnings indicating that the default peers could not be resolved. This is because the default peer list includes locators that depend on those transports.
- ❑ If you disable the UDPv4 transport but leave your multicast receive address unchanged, the middleware will log warnings indicating that the address could not be resolved. This is because the default multicast receive address depends on that transport.
- ❑ If you wish to disable the *Connection*'s use of a particular network transport, you must disable it at the transport level; it is not enough to simply remove uses of it from the peer list and multicast receive address.

Example: Use shared memory transport only

```
<connection_factory name="Example Factory">
    <transport_builtin>
            <mask>TRANSPORTBUILTIN_SHMEM</mask>
    </transport_builtin>
    <discovery>
            <initial_peers>
                    <element>shmem://</element>
            </initial_peers>
            <multicast_receive_addresses>
                            <!-- empty -->
            </multicast_receive_addresses>
    </discovery>
</connection_factory>
Example: Disable shared memory
<connection_factory name="Example Factory">
    <transport_builtin>
            <mask>TRANSPORTBUILTIN_UDPv4</mask>
    </transport_builtin>
    <discovery>
           <initial_peers>
                  <element>udpv4://239.255.0.1</element>
                  <element><-- optional: 4@... -->
                         udpv4://127.0.0.1
                  </element>
           </initial_peers>
    </discovery>
</connection_factory>
```

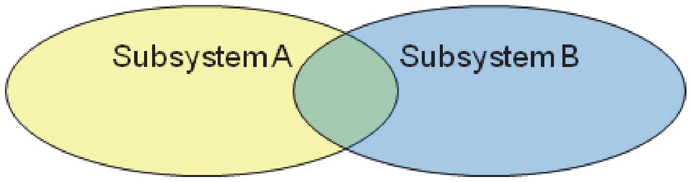
4.2.3 Asymmetric Discovery Configuration

In general, all applications on the same physical network and in the same domain are logically “intended” to communicate. In practice, however, you may find such a configuration undesirable. You may know, for example, that certain subsystems have very little need to exchange data, and you may wish to decrease your system’s bandwidth and/or memory usage and increase its scalability by limiting cross-talk.

You can design such a system using multiple domains, and route any traffic that must flow between them explicitly through a bridge: an application that joins both domains and publishes on one what it has received from the other. In some cases, you can also achieve similar results in a lighter-weight manner by manipulating the network addresses used by the middleware. This section will take you through two examples of such configurations.

4.2.3.1 Scenario: Overlapping Subsystems

Suppose your network contains two subsystems that, for the most part, do not need to exchange messages between them. However, there are a few applications that sit “in the middle” and need to send and receive messages to and from both subsystems. Building bridges could introduce performance bottlenecks and make the configuration more complex.



As an alternative, you can manipulate the addressing to achieve the same results without sacrificing your peer-to-peer architecture or introducing additional system components.

- ❑ Use one multicast address for communication within the first subsystem.
- ❑ Use a second multicast address to communication within the second subsystem.
- ❑ Configure applications in both subsystems to announce to both addresses.

Example: Subsystem A Configuration

```
<connection_factory name="A Factory">
    <discovery>
           <initial_peers>
                  <!-- Multicast address for subsystem A: -->
                  <element>239.255.0.1</element>
           </initial_peers>
           <multicast_receive_addresses>
                  <!-- Multicast address for subsystem A: -->
                  <element>239.255.0.1</element>
           </multicast_receive_addresses>
    </discovery>
</connection_factory>
```

Example: Subsystem B Configuration

```
<connection_factory name="B Factory">
    <discovery>
           <initial_peers>
                  <!-- Multicast address for subsystem B: -->
                  <element>239.255.0.2</element>
           </initial_peers>
           <multicast_receive_addresses>
                  <!-- Multicast address for subsystem B: -->
                  <element>239.255.0.2</element>
           </multicast_receive_addresses>
    </discovery>
</connection_factory>
```

Example: A+B Applications

```
<connection_factory name="AB Factory">
    <discovery>
           <initial_peers>
                  <!-- Multicast address for subsystem A: -->
                  <element>239.255.0.1</element>
                  <!-- Multicast address for subsystem B: -->
                  <element>239.255.0.2</element>
           </initial_peers>
           <multicast_receive_addresses>
```

```
                  <!-- Empty -->
           </multicast_receive_addresses>
    </discovery>
</connection_factory>
```

As you can see, the configurations above can be adapted to support any number of overlapping subsystems: simply choose more multicast addresses.

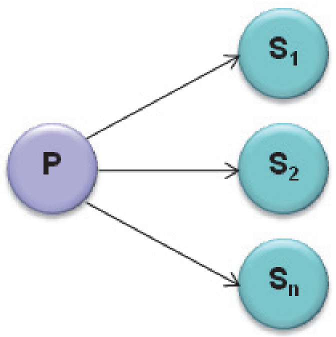
You may also notice that the A+B applications do not actually appear in the initial peer list of either the A or B applications. They will nevertheless communicate because of the ability of *RTI Message Service* to add new peers dynamically: when an A or B application receives an announcement from an A+B application, it will automatically add that application to its peer list and begin communicating with it. In this case, because the A+B applications do not specify a multicast receive address, communication in the A-to-A+B and B-to-A+B directions will take place using IP unicast or shared memory (since the shared memory transport was not turned off). If you expect most A+B applications to subscribe to the same topics, you may want to consider allocating an additional multicast address for use just by that overlapping group; use it as the multicast receive address of the AB Factory.

The dynamic peer addition capability described above is governed by the QoS parameter **accept_unknown_peers** in the Discovery QoS policy. This Boolean flag takes the value `true` by default, which is why it is not included in the XML above. The following configuration causes an application to communicate only with other applications that appear in its **initial_peers** list:

```
<connection_factory name="Example Factory">
    <discovery>
           <accept_unknown_peers>false</accept_unknown_peers>
    </discovery>
</connection_factory>
```

4.2.3.2 Scenario: One-Way Communication with High Fan-Out

In some systems, a single publisher (or a small number of publishers) distribute(s) messages to a very large number of subscribers. These subscribers do not exchange data with each other, so their performance can potentially be improved and their memory footprint decreased by avoiding that part of the discovery process.



To configure the system in this way, do the following:

- ❑ *Use IP multicast to distribute messages from the publisher to the subscribers.* The publisher sends, and subscribers receive, messages to and from this address. Traffic on this address does not flow the in the other direction.
- ❑ *Use IP unicast to send acknowledgements* and any other subscriber-to-publisher back channel messages. Unicast is appropriate in this case because of the small number of recipients (possibly only one); it ensures that the network interfaces and CPUs on the other subscribing machines will not be burdened with unnecessary traffic.

Because the subscribers do not send their own discovery declarations to any address on which they listen, they will never discovery each other.

Example: Publisher configuration

```
<connection_factory name="Publisher Factory">
    <discovery>
           <initial_peers>
                  <!-- Multicast address to talk to subscribers: -->
                  <element>239.255.0.1</element>
           </initial_peers>
           <multicast_receive_addresses>
                         <!-- Empty: only listen over unicast -->
           </multicast_receive_addresses>
    </discovery>
</connection_factory>
```

Example: Subscriber configuration

```
<connection_factory name="Subscriber Factory">
    <discovery>
           <initial_peers>
                  <!-- Publisher's unicast address -->
                  <element>192.168.1.100</element>
                  <!-- More addresses if other (potential) publish-
ers: ... -->
           </initial_peers>
           <multicast_receive_addresses>
                  <!-- Multicast address to talk to publisher: -->
                  <element>239.255.0.1</element>
           </multicast_receive_addresses>
    </discovery>
</connection_factory>
```

If storing particular nodes' unicast addresses in the configuration file represents a configuration management challenge for you, consider one of these alternatives:

❑ *Use a second multicast address* for subscriber-to-publisher (meta-) traffic instead of unicast addresses. Set this address as the publisher's multicast receive address and add it to the subscriber's initial peers. This configuration also has the benefit of requiring no changes if the number of publisher's increases. One caveat: multicast traffic is typically more expensive for the operating system's network stack to process than unicast traffic, so the subscribers' responsiveness to the publisher could be impacted. This penalty may or may not be noticeable in your system, however.

- ❑ *Select the unicast address at runtime* using an environment variable, a per-node configuration file, a communication back channel, or some other mechanism. Use this address to override the RTI configuration file using the runtime mechanism described Chapter 2, "Connecting to the Network," in the User's Manual.

4.2.4 Discovery Implementation

Advanced

Note: This section contains advanced material not required by most users.

4.2.4.1 Connection Discovery

Let's examine what happens when a new remote *Connection* is discovered. To summarize Connection Discovery (Section 4.2.1.1):

- ❑ Once a remote connection has been added to the *RTI Message Service* internal database, *RTI Message Service* keeps track of that remote connection's **connection_liveliness_lease_duration**. If a declaration from that connection is not received at least once within the **connection_liveliness_lease_duration**, the remote connection is considered stale, and the remote connection, together with all its entities, will be removed from the database of the local connection.
- ❑ To keep from being purged by other connections, each connection needs to periodically send a declaration to refresh its liveliness. The rate at which these declarations are sent is controlled by the **connection_liveliness_assert_period** in the connection's Discovery Config QoS policy. This exchange, which keeps Connection A from appearing 'stale,' is illustrated in Figure 4.2, "Periodic 'connection DATAs'," on page 4-22.

  Figure 4.3, "Ungraceful Termination of a Connection," on page 4-23 shows what happens when Connection A terminates ungracefully and therefore needs to be seen as 'stale.'

- ❏ The **connection_liveliness_assert_period, connection_liveliness_lease_duration**, **min_initial_connection_announcement_period** and **max_initial_connection_announcement_period** can be set as follows:

```
<connection_factory name="A">
    <discovery_config>
        <connection_liveliness_assert_period>
                <sec>1</sec>
                <nanosec>0</nanosec>
            </connection_liveliness_assert_period>
        <connection_liveliness_lease_duration>
            <sec>10</sec>
            <nanosec>0</nanosec>
        </connection_liveliness_lease_duration>
        <!-- Default values: -->
        <min_initial_connection_announcement_period>
            <sec>1</sec>
            <nanosec>0</nanosec>
        </min_initial_connection_announcement_period>
        <max_initial_connection_announcement_period>
            <sec>1</sec>
            <nanosec>0</nanosec>
        </max_initial_connection_announcement_period>
    </discovery_config>
</connection_factory>
```

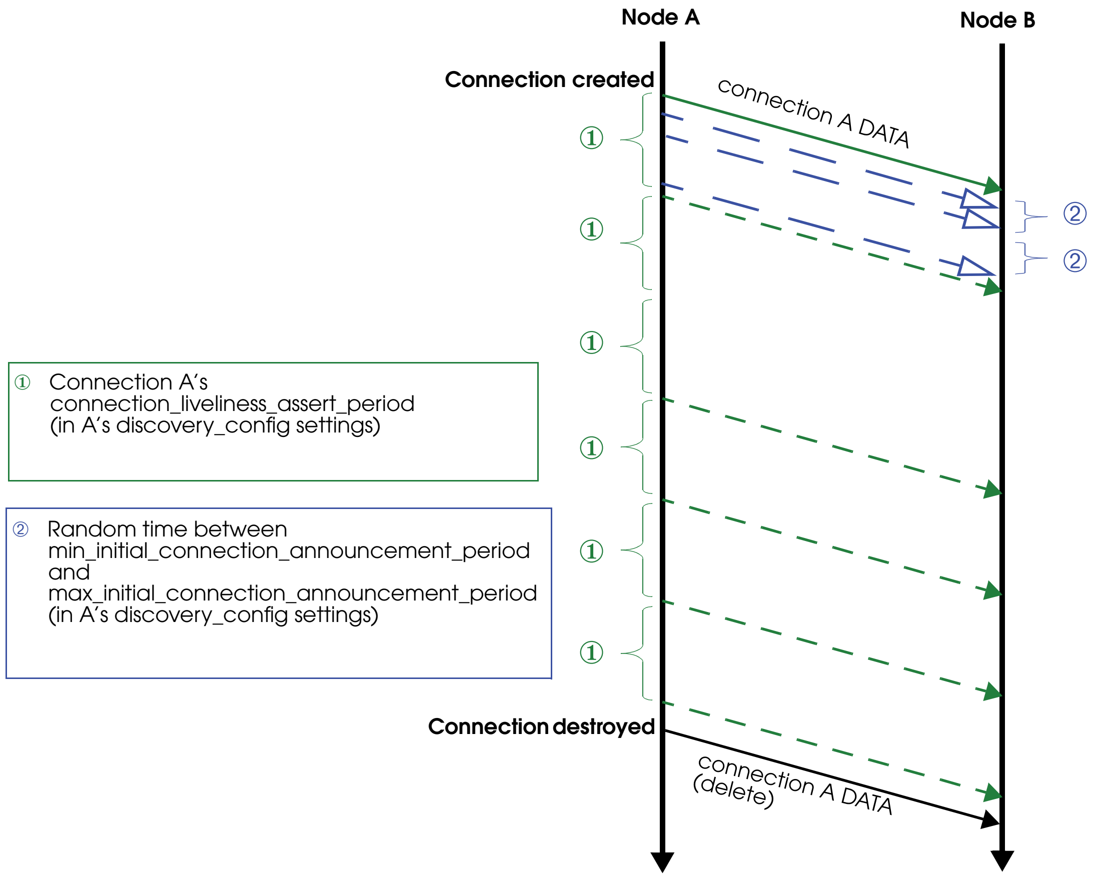
4.2.4.2 Endpoint Discovery

When you create a *MessageProducer*/*MessageConsumer* for your user data, a publication/ subscription declaration describing the newly created object is sent from the local discovery endpoint producer to the remote *Connection*s that are currently in the local database.

Similarly, if the application deletes any producers/consumers, the connection sends publication/subscription deletion declarations.

When a remote entity record is added or removed in the database, matching is performed with all the local entities. Only after there is a successful match on both ends can an application's user-created *MessageConsumers* and *MessageProducers* communicate with each other.

Figure 4.2 **Periodic 'connection DATAs'**

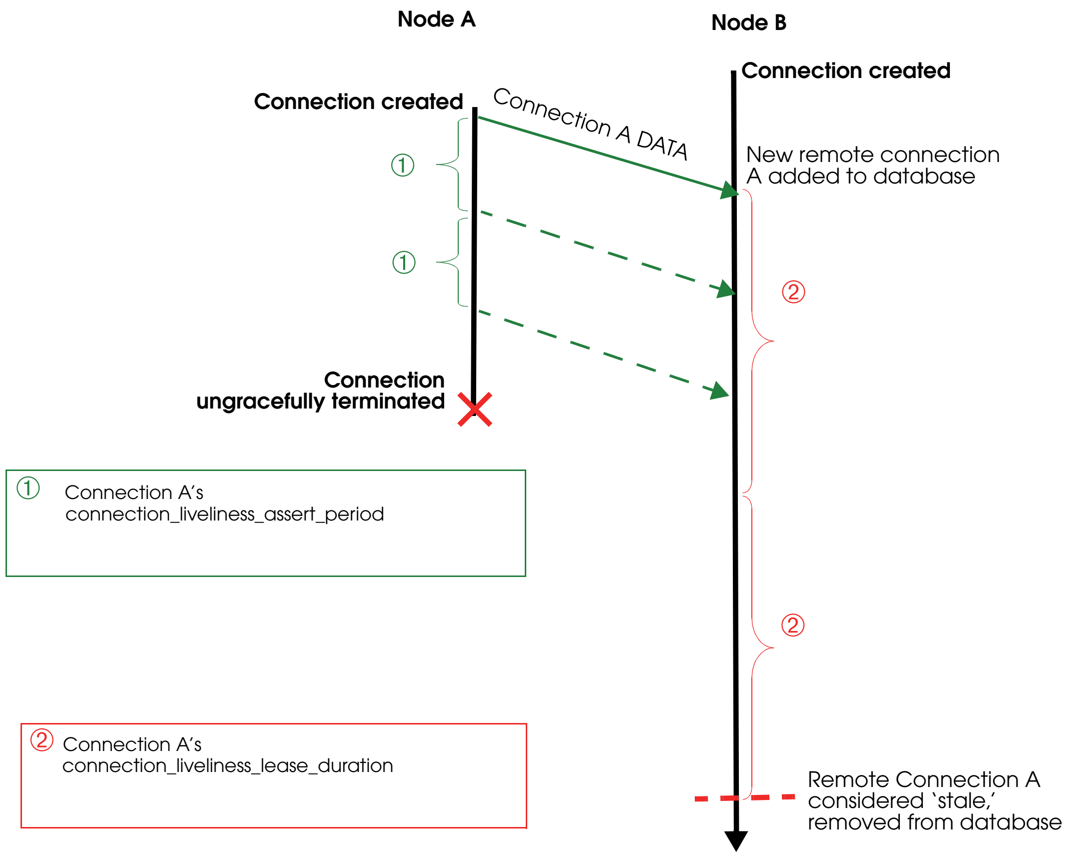

Node A
Node B
Connection created
connection A DATA
①
②
① Connection A's connection_liveliness_assert_period (in A's discovery_config settings)
② Random time between min_initial_connection_announcement_period and max_initial_connection_announcement_period (in A's discovery_config settings)
Connection destroyed
connection A DATA (delete)

The Connection on Node A sends a 'connection DATA' to Node B, which is in Node A's peers list. This occurs regardless of whether or not there is an RTI Message Service application on Node B.

① *The green short dashed lines are periodic connection DATAs. The time between these messages is controlled by the connection_liveliness_assert_period (in A's <discovery_config> settings).*

② *In addition to the periodic connection DATAs, 'initial repeat messages' (shown in blue, with longer dashes) are sent from A to B. These messages are sent at a random time between min_initial_connection_announement_period and max_initial_connection_announcement_period (in A's <discovery_config> settings). The number of these initial repeat messages is set in initial_connection_announcements.*

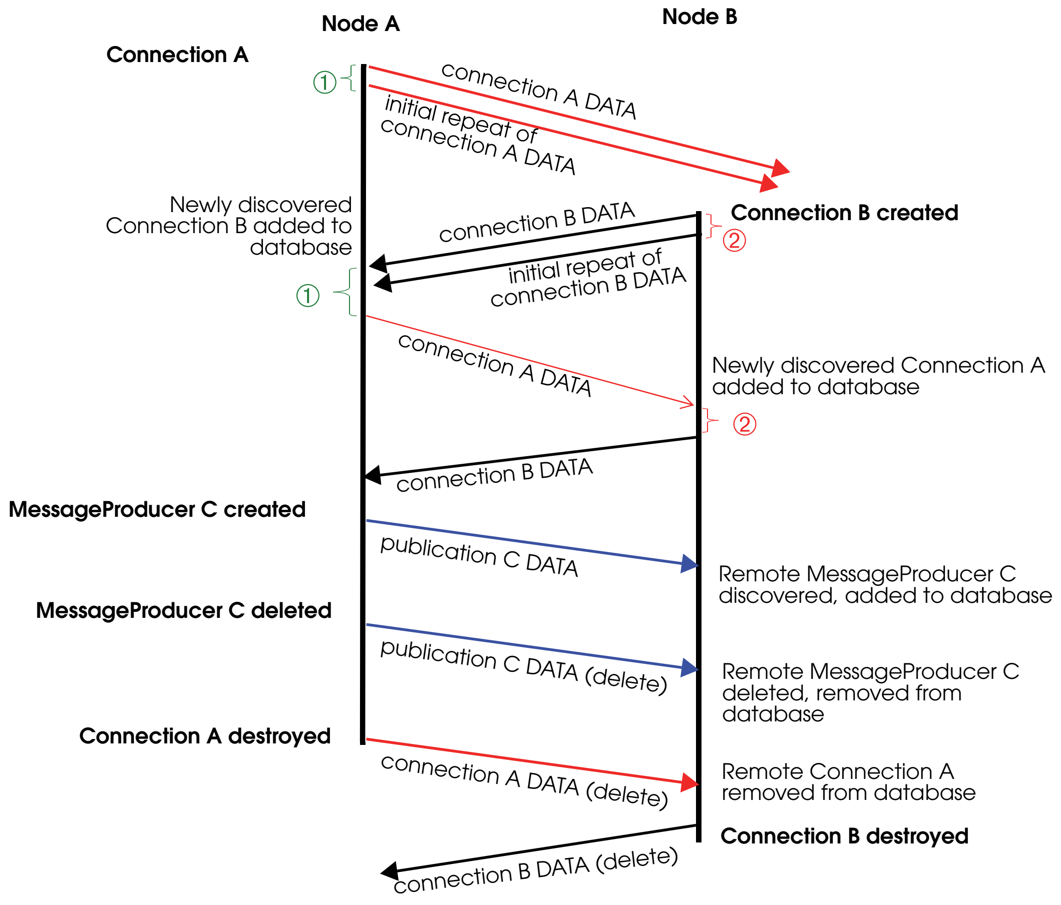
Figure 4.3 **Ungraceful Termination of a Connection**


Node A
Node B
Connection created
Connection created
Connection A DATA
New remote connection A added to database
①
①
②
②
Connection ungracefully terminated
① Connection A's connection_liveliness_assert_period
② Connection A's connection_liveliness_lease_duration
Remote Connection A considered 'stale,' removed from database

Connection A is removed from Connection B's database if it is not refreshed within the liveliness lease duration. Dashed lines are periodic connection DATA messages.

(Periodic resends of 'connection B DATA' from B to A are omitted from this diagram for simplicity. Initial repeat messages from A to B are also omitted from this diagram—these messages are sent at a random time between min_initial_connection_announcement_period and max_initial_connection_announcement_period.)

4.2.4.3 Discovery Traffic Summary


Node A
Node B
Connection A
①
connection A DATA
initial repeat of connection A DATA
Connection B created
②
Newly discovered Connection B added to database
connection B DATA
initial repeat of connection B DATA
①
connection A DATA
Newly discovered Connection A added to database
②
connection B DATA
MessageProducer C created
publication C DATA
Remote MessageProducer C discovered, added to database
MessageProducer C deleted
publication C DATA (delete)
Remote MessageProducer C deleted, removed from database
Connection A destroyed
connection A DATA (delete)
Remote Connection A removed from database
Connection B destroyed
connection B DATA (delete)

① wait random time (between min_initial_connection_announcement_period and max_initial_connection_announcement_period) for (initial_connection_announcements) = 1 (using values for connection A)

② same as ①, but using connection B's settings

This diagram shows both phases of the discovery process. Connection A is created first, followed by Connection B. Each has the other in its peers list. After they have discovered each other, a MessageProducer is created on Connection A. Periodic connection DATAs, HBs and ACK/NACKs are omitted from this diagram.

4.2.5 Debugging Discovery

To understand the flow of messages during discovery, you can increase the verbosity of the messages logged by *RTI Message Service* so that you will see whenever a new object is discovered and whenever there is a match between a local entity and a remote entity.

This can be achieved with the logging API:

```
com.rti.management.Logger.setVerbosityByCategory(
    com.rti.management.Logger.Category.ENTITIES,
    com.rti.management.Logger.Verbosity.STATUS_REMOTE);
```

Using the scenario in the summary diagram in section Discovery Traffic Summary (Section 4.2.4.3), these are the messages as seen on Connection A:

```
[D0049|ENABLE]DISCPluginManager_onAfterLocalParticipantEnabled:annou
ncing new local participant: 0XA0A01A1,0X5522,0X1,0X1C1
[D0049|ENABLE]DISCPluginManager_onAfterLocalParticipantEnabled:at
{46c614d9,0C43B2DC}
```

(The above messages mean: First connection A declaration sent out when connection A is enabled.)

```
DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:d
iscovered new participant: host=0x0A0A01A1, app=0x0000552B,
instance=0x00000001
DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:a
t {46c614dd,8FA13C1F}
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin dis-
covered/updated remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at
{46c614dd,8FACE677}
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin
accepted new remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at
{46c614dd,8FACE677}
```

(The above messages mean: Received connection B declaration.)

```
DISCSimpleParticipantDiscoveryPlugin_remoteParticipantDiscovered:re-
announcing participant self: 0XA0A01A1,0X5522,0X1,0X1C1
DISCSimpleParticipantDiscoveryPlugin_remoteParticipantDiscovered:at
{46c614dd,8FC02AF7}
```

(The above messages mean: Resending connection A declaration to the newly discovered remote connection.)

```
PRESPsService_linkToLocalReader:assert remote
0XA0A01A1,0X552B,0X1,0X200C2, local 0x000200C7 in reliable reader
service
PRESPsService_linkToLocalWriter:assert remote
0XA0A01A1,0X552B,0X1,0X200C7, local 0x000200C2 in reliable writer
service
PRESPsService_linkToLocalWriter:assert remote
0XA0A01A1,0X552B,0X1,0X4C7, local 0x000004C2 in reliable writer ser-
vice
PRESPsService_linkToLocalWriter:assert remote
0XA0A01A1,0X552B,0X1,0X3C7, local 0x000003C2 in reliable writer ser-
vice
PRESPsService_linkToLocalReader:assert remote
0XA0A01A1,0X552B,0X1,0X4C2, local 0x000004C7 in reliable reader ser-
vice
PRESPsService_linkToLocalReader:assert remote
0XA0A01A1,0X552B,0X1,0X3C2, local 0x000003C7 in reliable reader ser-
vice
PRESPsService_linkToLocalReader:assert remote
0XA0A01A1,0X552B,0X1,0X100C2, local 0x000100C7 in best effort reader
service
```

(The above messages mean: Automatic matching of the discovery consumers and producers. A built-in remote endpoint's object ID always ends with Cx.)

```
DISCSimpleParticipantDiscoveryPluginReaderListener_onDataAvailable:d
iscovered modified participant: host=0x0A0A01A1, app=0x0000552B,
instance=0x00000001
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:plugin dis-
covered/updated remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_assertRemoteParticipant:at
{46c614dd,904D876C}
```

(The above messages mean: Received connection B declaration.)

```
DISCPluginManager_onAfterLocalEndpointEnabled:announcing new local
publication: 0XA0A01A1,0X5522,0X1,0X80000003
DISCPluginManager_onAfterLocalEndpointEnabled:at {46c614d9,1013B9F0}
DISCSimpleEndpointDiscoveryPluginPDFListener_onAfterLocalWriterEnabl
ed:announcing new publication: 0XA0A01A1,0X5522,0X1,0X80000003
DISCSimpleEndpointDiscoveryPluginPDFListener_onAfterLocalWriterEnabl
ed:at {46c614d9,101615EB}
```

(The above messages mean: Publication C declaration has been sent.)

```
DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAv
ailable:discovered subscription: 0XA0A01A1,0X552B,0X1,0X80000004
DISCSimpleEndpointDiscoveryPlugin_subscriptionReaderListenerOnDataAv
ailable:at {46c614dd,94FAEFEF}
DISCEndpointDiscoveryPlugin_assertRemoteEndpoint:plugin discovered/
updated remote endpoint: 0XA0A01A1,0X552B,0X1,0X80000004
DISCEndpointDiscoveryPlugin_assertRemoteEndpoint:at
{46c614dd,950203DF}
```

(The above messages mean: Receiving subscription D declaration from Node B.)

```
PRESPsService_linkToLocalWriter:assert remote
0XA0A01A1,0X552B,0X1,0X80000004, local 0x80000003 in best effort
writer service
```

(The above message means: User-created MessageProducer C and MessageConsumer D are matched.)

```
[D0049|DELETE_CONTAINED]DISCPluginManager_onAfterLocalEndpointDelete
d:announcing disposed local publication:
0XA0A01A1,0X5522,0X1,0X80000003
[D0049|DELETE_CONTAINED]DISCPluginManager_onAfterLocalEndpointDelete
d:at {46c61501,288051C8}
[D0049|DELETE_CONTAINED]DISCSimpleEndpointDiscoveryPluginPDFListener
_onAfterLocalWriterDeleted:announcing disposed publication:
0XA0A01A1,0X5522,0X1,0X80000003
[D0049|DELETE_CONTAINED]DISCSimpleEndpointDiscoveryPluginPDFListener
_onAfterLocalWriterDeleted:at {46c61501,28840E15}
```

(The above messages mean: Publication C declaration(delete) has been sent.)

```
DISCPluginManager_onBeforeLocalParticipantDeleted:announcing before
disposed local participant: 0XA0A01A1,0X5522,0X1,0X1C1
DISCPluginManager_onBeforeLocalParticipantDeleted:at
{46c61501,28A11663}
```

(The above messages mean: Connection A declaration(delete) has been sent.)

```
DISCParticipantDiscoveryPlugin_removeRemoteParticipantsByCookie:plug
in removing 3 remote entities by cookie
DISCParticipantDiscoveryPlugin_removeRemoteParticipantsByCookie:at
{46c61501,28E38A7C}
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:plugin dis-
covered disposed remote participant: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:at
{46c61501,28E68E3D}
```

```
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:remote
entity removed from database: 0XA0A01A1,0X552B,0X1,0X1C1
DISCParticipantDiscoveryPlugin_removeRemoteParticipantI:at
{46c61501,28E68E3D}
```

(The above messages mean: Removing discovered entities from local database, before shutting down.)

As you can see, the messages are encoded, since they are primarily used by RTI support personnel.

If a remote entity is discovered, but does not match with a local entity as expected, check the QoS of both the remote and local entity.

4.3 Tune Reliability Performance

You can tune the behavior of the reliability protocol to create the right balance between performance and "chattiness" for your application. This fine-grained configuration is not necessary for all applications, and a misconfiguration can lead to poor performance or a loss of reliability, so only continue with this section if you know that the default behavior is not meeting your needs.

Before you can tune the protocol, you will need to understand how it works.

4.3.1 Introduction to the Reliability Protocol

The reliability protocol consists of two primary metadata messages, which are sent and received internally by the middleware.

❏ A message producer sends *heartbeats* to its message consumers. These identify the range of historical messages—identified by sequence number—that the producer currently has available.

❏ A message consumer responds to a heartbeat with an *acknowledgement* message indicating the messages it has received. (The same RTPS message functions as both a positive and negative acknowledgement ("ACK" and "NACK"). However, ACK and NACK are sometimes colloquially discussed as if they were separate messages, depending on whether or not the consumer is up-to-date.)

In order to achieve maximum determinism and make your tuning job easier, the reliability protocol is almost entirely driven by the message producer. A message consumer will send an ACK/NACK under only two conditions:

- ❑ **It has just matched with a new producer**. To encourage the producer to send it historical data as quickly as possible, it will send the producer a "zero-ACK": an ACK/NACK message indicating no previously received messages.
- ❑ **It has received a heartbeat from a producer**. It will respond to the heartbeat with an ACK/NACK indicating its progress.

  A consumer can also be configured to indicate negative acknowledgement only—that is, to respond to heartbeats only when it has not received all messages. This behavior sacrifices some degree of reliability in exchange for improved performance in topologies with many consumers for each producer.

Except when it matches a producer for the first time, a consumer will never send an ACK/NACK of its own volition.

Similarly, a producer sends historical data—for durability or repair purposes—only upon receipt of a NACK. It will not preemptively send such data.

Message producers send heartbeats in two ways:

- ❑ **Periodically**: A producer sends heartbeats at an application-configurable rate. This rate will adjust according to the circumstances:
- ❑ The *late-joiner heartbeat rate* is faster than the steady-state heartbeat rate in order to bring subscribers up-to-date as quickly as possible.
- ❑ The *fast heartbeat rate* applies when the producer's cache of historical data is near full in order to help slow consumers to catch up more quickly, thereby allowing the producer to empty its cache and avoid blocking.
- ❑ The *normal heartbeat rate* applies in steady state, when the other rates do not.

  Periodic heartbeats are important to maintaining reliability in the case where (*1*) a message is lost and (*2*) another message will not be sent for some time.
- ❑ **"Piggybacked" on application messages**: Every few messages, a producer will place a heartbeat into the same network packet as an application message. These piggyback heartbeats are important to maintaining the responsiveness of the reliability protocol, because they allow consumers to respond immediately upon realizing that they have missed a previous message. (Recall that consumers send NACKs only in response to heartbeats.)

4.3.2 Configuring Heartbeats

Heartbeats are configured on a per-*Topic* basis, using the *MessageProducer* Protocol QoS policy.

4.3.2.1 Periodic Heartbeats

As described above, there are three heartbeat rates; which one is in force depends on the circumstance.

- ❏ When a new matching message consumer is discovered after the producer has published some messages, the producer will communicate with it using the *late-joiner heartbeat period*; this period must be faster (or equal to) the normal heartbeat period in order to help the new consumer catch up. Once the new consumer has caught up, the producer will return to the *normal heartbeat rate*.
- ❏ When the number of unacknowledged messages in the producer's historical cache surpasses a specified *high watermark*, the producer switches to the *fast heartbeat period*. This faster period encourages consumers to acknowledge messages faster, thereby allowing the producer to clear out its historical cache and make room for new messages. If the producer can maintain room in its cache for new messages, it will not need to block when sending a new message, which would impact throughput and latency.
- ❏ When a producer is using its fast heartbeat period, and sufficient acknowledgements arrive to bring the number of unacknowledged messages in its cache below a specified *low watermark*, the producer will switch back to its *normal heartbeat period*.

The following configuration example shows the default values of these parameters:

```
<topic name="Example Topic">
    <producer_defaults>
           <protocol>
                  <rtps_reliable_producer>
                         <low_watermark>0</low_watermark>
                         <high_watermark>1</high_watermark>
                         <heartbeat_period>
                                <sec>3</sec>
                                <nanosec>0</nanosec>
                         </heartbeat_period>
                         <fast_heartbeat_period>
                                <sec>3</sec>
                                <nanosec>0</nanosec>
                         </fast_heartbeat_period>
                         <late_joiner_heartbeat_period>
                                <sec>3</sec>
                                <nanosec>0</nanosec>
                         </late_joiner_heartbeat_period>
                  </rtps_reliable_producer>
           </protocol>
    </producer_defaults>
</topic>
```

The middleware will issue notifications when the high and low watermarks are crossed so that the application can monitor the middleware's performance.

Table 4.4 **Notification Type: StatusNotifier.RELIABLE_PRODUCER_CACHE_CHANGED_NOTIFICATION_TYPE**

The number of unacknowledged messages in a reliable message producer's cache has changed: the cache is empty, full, or has just crossed a high or low watermark.

Attribute Name	Attribute Type	Description
unacknowledgedMessageCount	int	The number of messages in the producer's cache that has not been acknowledged by at least one consumer.
unacknowledgedMessageCountPeak	int	The highest value that **unacknowledgedMessageCount** has reached thus far.

See Chapter 2, "Connecting to the Network," in the User's Manual for information about how to receive status notifications.

4.3.2.2 Piggyback Heartbeats

A piggyback heartbeat is a heartbeat that is embedded in the same network packet as an application data message instead of being sent separately. It is functionally identical to a periodic heartbeat; the only difference is when it is sent.

The frequency with which a heartbeat is added to a message's packet is defined based on the size of the message producer's cache. The frequency is called *heartbeats per max_messages*; the following XML shows the default value:

```
<topic name="Example Topic">
    <producer_defaults>
            <protocol>
                    <rtps_reliable_producer>
                            <heartbeats_per_max_messages>
                                    8
                            </heartbeats_per_max_messages>
                    </rtps_reliable_producer>
            </protocol>
    </producer_defaults>
</topic>
```

The term "max messages" here means one of two things:

- ❑ **If batching is disabled** (this is the default setting; see Chapter 5: Throughput Management for more information about this feature): "max messages" refers to the **max_messages** field of the Resource Limits QoS policy. This policy is described in the Chapter 6, "Scalable High-Performance Applications: Message Reliability," in the User's Manual.
- ❑ **If batching is enabled**: "max messages" refers to the **max_batches** field of the *MessageProducer* Resource Limits QoS policy. This policy is described Section 5.1.1.2, Batching and Reliability, in the User's Manual.

If **heartbeats_per_max_messages** is set to zero, no piggyback heartbeat will be sent. If **max_messages** (or **max_batches**, as appropriate) is set to LENGTH_UNLIMITED, 100 million is assumed for the purpose of this parameter.

4.3.2.3 Configuring Discovery Reliability

The middleware's internal endpoint—producer and consumer—discovery communication channels are reliable, and you can configure their acknowledgement behavior just as you can configure your own message producer. These discovery settings are specified per-*ConnectionFactory* within the **publication_producer** and **subscription_producer** elements of the Discovery Config QoS policy, which have the same internal structure as the **rtps_reliable_producer** elements shown above.

For example:

```
<connection_factory name="Example Factory">
    <discovery_config>
            <publication_producer>
                    <heartbeats_per_max_messages>
                            8
                    </heartbeats_per_max_messages>
                    <low_watermark>0</low_watermark>
                    <high_watermark>1</high_watermark>
                    <heartbeat_period>
                            <sec>3</sec>
                            <nanosec>0</nanosec>
                    </heartbeat_period>
                    <fast_heartbeat_period>
                            <sec>3</sec>
                            <nanosec>0</nanosec>
                    </fast_heartbeat_period>
                    <late_joiner_heartbeat_period>
                            <sec>3</sec>
                            <nanosec>0</nanosec>
                    </late_joiner_heartbeat_period>
            </publication_producer>
            <subscription_producer>
                    <heartbeats_per_max_messages>
                            8
                    </heartbeats_per_max_messages>
                    <low_watermark>0</low_watermark>
                    <high_watermark>1</high_watermark>
                    <heartbeat_period>
                            <sec>3</sec>
                            <nanosec>0</nanosec>
                    </heartbeat_period>
                    <fast_heartbeat_period>
                            <sec>3</sec>
                            <nanosec>0</nanosec>
                    </fast_heartbeat_period>
                    <late_joiner_heartbeat_period>
                            <sec>3</sec>
                            <nanosec>0</nanosec>
                    </late_joiner_heartbeat_period>
            </subscription_producer>
    </discovery_config >
</topic>
```

4.3.3 Configuring Acknowledgements

By default, a *MessageConsumer* will respond to heartbeats regardless of whether it is "caught up," sending ACKs or NACKs as appropriate. This configuration provides the highest degree of reliability, as it ensures that the *MessageProducer* will purge messages from its cache only when it has received positive acknowledgement of the receipt of those messages from all of its matched consumers.

However, the default configuration may not be appropriate in all cases. When there are many consumers matched to a single producer, and the underlying network is highly reliable, the vast majority of heartbeat responses will indicate that all messages have been received—that is, that no further action on the producer's part is necessary—and yet the great number of these responses will place a burden on the system. To improve performance in such cases, you may want to disable positive acknowledgements, that is, to configure the consumers to respond to heartbeats only when they have a missed message to report.

Consumers will automatically report their acknowledgement settings to their matched producers. For consumers with positive acknowledgements disabled, a producer will retain sent messages for a *keep duration*, after which, if no negative acknowledgements have been received, it will discard the message.

A single producer can communicate with many consumers with different acknowledgement configurations. In a mixed configuration, it will discard a sent message only after (*a*) all positively acknowledging consumers have ACKed the message *and* (*b*) the keep duration has elapsed for all non-positively acknowledging consumers.

4.3.3.1 Disabling Positive Acknowledgement for a MessageConsumer

By default, a *MessageConsumer* will respond to heartbeats with both positive and negative acknowledgements, as appropriate. To disable positive acknowledgements, use the *MessageConsumer* Protocol QoS policy as shown:

```
<topic name="Example Topic">
    <consumer_defaults>
        <protocol>
           <disable_positive_acks>true</disable_positive_acks>
        </protocol>
    </consumer_defaults>
</topic>
```

4.3.3.2 Modifying the *MessageProducer*'s Keep Duration

A *MessageProducer* uses an adaptive algorithm to determine how long to retain sent messages for non-positively acknowledging consumers.

❑ At first, it will keep a sent message for the *disable positive ACKs minimum message keep duration*. If no NACKs are received within that amount of time, the message will be discarded.

❑ If one or more NACKs are received within the keep duration—most likely indicating network congestion—the producer will adapt to the level of congestion by gradually increasing the keep duration up to the *disable positive ACKs maximum keep duration*. In this way, it will effectively throttle its own send rate to maximize throughput while minimizing the number of dropped messages.

❑ If the level of congestion decreases, the producer will gradually decrease its keep duration again, improving throughput as network conditions improve.

The following XML shows the default values of these two parameters:

```
<topic name="Example Topic">
    <producer_defaults>
        <protocol>
            <rtps_reliable_producer>
                <disable_positive_acks_min_message_keep_duration>
                    <sec>0</sec>
                    <nanosec>1000000<!-- 1 ms --></nanosec>
                </disable_positive_acks_min_message_keep_duration>
                <disable_positive_acks_max_message_keep_duration>
                    <sec>1</sec>
                    <nanosec>0</nanosec>
                </disable_positive_acks_max_message_keep_duration>
            </rtps_reliable_producer>
        </protocol>
    </producer_defaults>
</topic>
```

Chapter 5 Throughput Management

You want to optimize your application's throughput. At the same time, you need to ensure that your network resources are not overwhelmed. If a traffic surge does occur, you need to take care that your application responds with agility and robustness to avoid NACK storms and a loss of connectivity. *RTI Message Service* can help.

This chapter covers two primary aspects of your data throughput:

- ❏ Maximizing the message throughput of your application, including how to tune the middleware and how to avoid problems with slow message consumers.
- ❏ Managing surges in network traffic that could impede your messages, including how to avoid and respond to NACK storms.

This chapter includes the following sections:

5.1 Maximizing Throughput

This section addresses two aspects of increasing throughput: increasing the efficiency of message transmission to boost throughput and preventing consumers that can't keep up from slowing down the entire system.

5.1.1 Batch Messages to Increase Throughput

If your application sends relatively small—less than a kilobyte or two—messages at a very high rate, you may find that the network itself becomes a bottleneck. The time it takes for each packet to traverse the network stack, and the time it takes for acknowledgements to return, may result in under-utilization of the network's bandwidth.

RTI Message Service supports *batching* messages at the network level to allow the overhead of packet headers, the cost of system calls, and the subscriber-side CPU burden of sending acknowledgements to be amortized across a large number of messages. By combining multiple messages into a single network packet in this way, the middleware can potentially increase throughput many-fold.

Batching is often appropriate when your application sends a large number of messages at a high rate. In other scenarios, it may not be. When your application sends a message, and the middleware is building a batch, it will not put that message on the network immediately. Instead, it will hold that message and wait to accumulate additional messages before sending them all at once. If there is a significant pause in between when your application sends consecutive messages, this accumulation time—which translates directly into end-to-end latency—may grow unacceptably long. You can bound this latency by sending accumulated batched messages on a timer (see below), but if the batches have only accumulated one or two messages by this time, you may see little performance benefit.

5.1.1.1 Building and Sending Batches

Turning on batching is easy:

```
<topic name="Example Topic">
    <producer_defaults>
            <batch>
                    <enable>true</enable>
            </batch>
    </producer_defaults>
</topic>
```

A message consumer requires no special configuration in order to receive and deliver batches. Batching is transparent on the subscribing side.

Once the middleware has begun building a batch, it will *flush* that batch—that is, send it on the network—under three conditions. The XML below shows the default values of the highlighted QoS parameters.

When the batch's size reaches a certain **number of messages**:	`<topic name="SampleTopic"> <producer_defaults> <batch> <enable>true</enable> <max_messages> LENGTH_UNLIMITED </max_messages> </batch> </producer_defaults> </topic>`
When the batch's size reaches a certain **number of bytes**:	`<topic name="SampleTopic"> <producer_defaults> <batch> <enable>true</enable> <max_data_bytes> 1024 </max_data_bytes> </batch> </producer_defaults> </topic>`
When the **elapsed time** exceeds a certain threshold:	`<topic name="SampleTopic"> <producer_defaults> <batch> <enable>true</enable> <max_flush_delay> <sec> DURATION_INFINITE_SEC </sec> <nanosec> DURATION_INFINITE_NANOSEC </nanosec> </max_flush_delay> </batch> </producer_defaults> </topic>`
Manually:	`void com.rti.RTIMessageProducer.flush() throws javax.jms.JMSException`

Example: Flush automatically based on several criteria

With the following configuration, the middleware will flush batches automatically any time one of the following conditions becomes true:

1. The batch has accumulated 100 messages
2. The batch has grown in size of 4 KB.
3. Half a second has elapsed since the last flush.

```
<topic name="SampleTopic">
    <producer_defaults>
            <batch>
                    <enable>true</enable>
                    <max_messages>100</max_messages>
                    <max_data_bytes>4096</max_data_bytes>
                    <max_flush_delay>
                            <sec>0</sec>
                            <nanosec>500000000</nanosec>
                    </max_flush_delay>
            </batch>
    </producer_defaults>
</topic>
```

Example: Flush manually

In this example, the middleware's automatic flushing mechanism is configured to operate only rarely. Instead, the application relies on manually flushing a group of messages each time it sends them.

In the configuration file:

```
<topic name="SampleTopic">
    <producer_defaults>
            <batch>
                    <enable>true</enable>
                    <max_data_bytes>32768</max_data_bytes>
            </batch>
    </producer_defaults>
</topic>
```

In application code:

```
Message msg1 = …;
Message msg2 = …;
Message msg3 = …;
MessageProducer pub = …;

pub.send(msg1);
pub.send(msg2);
pub.send(msg3);
((RTIMessageProducer) pub).flush();
```

5.1.1.2 Batching and Reliability

Reliability is carried out at the level of an entire batch, not an individual message within that batch. By allowing subscribers to issue a single ACK/NACK for an entire batch of messages, and publishers to maintain state at the granularity of a whole batch, the reliability protocol can operate much more efficiently and with lower overhead.

The **heartbeats_per_max_messages** parameter, introduced in Chapter 4: Data Connectivity, is interpreted in a different way when batching is enabled. Because reliability is at the level of a batch, only a single piggyback heartbeat will ever be attached to a batch. Therefore, "heartbeats per max messages" is really "heartbeats per max batches," where the maximum number of messages is configured with the following QoS parameter, shown below with its default value:

```
<topic name="SampleTopic">
   <producer_defaults>
        <producer_resource_limits>
             <max_batches>
                  LENGTH_UNLIMITED
             </max_batches>
        </producer_resource_limits>
   </producer_defaults>
</topic>
```

As described in Chapter 4: Data Connectivity, the sentinel LENGTH_UNLIMITED is considered equal to 100 million for the purposes of calculating how often the middleware will send piggyback heartbeats. For example, the default settings will lead to only very infrequent piggyback heartbeats: **max_batches** is equivalent to 100 million, and **heartbeats_per_max_samples** is 8, so a piggyback heartbeat will be attached to every 12.5 million batches. This frequency is not enough to significantly impact reliability behavior, even for very high-throughput systems; reliability instead relies on periodic heartbeats.

Other reliability-related resource limits—such as **max_messages** in the Resource Limits QoS policy introduced in Chapter 6, "Scalable High-Performance Applications: Message Reliability," in the User's Manual—remain unchanged in their interpretation. They always apply to individual samples, not to entire batches.

5.1.2 Dealing with Slow Consumers

Unfortunately, problems can occur if one or more consumers are not able to respond to the producer in a timely manner. If a producer's cache is full and it has not received a response from a particular consumer, it has only a few choices:

- ❏ **Don't expect acknowledgements** in the first place. You can configure your consumers to not provide positive acknowledgements when they receive messages, just negative acknowledgements when they *don't* receive something. This technique efficiently isolates the producer from slow consumers, but is only appropriate when the producer and consumer are loosely coupled and very strict reliability is not required. See Chapter 4: Data Connectivity for more information about this configuration.
- ❏ **Enlarge the cache.** This tactic can be a good one initially, but cannot continue indefinitely. See Chapter 6, "Scalable High-Performance Applications: Message Reliability," in the User's Manual for more information about producer cache size management and its relationship to reliability.
- ❏ **Make room in the cache by discarding messages** that have not yet been fully acknowledged. This action puts reliable delivery at risk for all other consumers, because if a consumer later NACKs a discarded message, the producer will be unable to repair the missing data.
- ❏ **Stop waiting for acknowledgements** from the slow consumer. Doing so may amount to failing the consumer over to a best-effort mode—simply not waiting for acknowledgment before flushing sent data from the queue—or, even more severe, refraining from sending future messages to the consumer altogether. This tactic puts reliability at risk, but only for the offending consumer(s).

This section summarizes and cross references material from other chapters in this manual and the User's Manual to provide a comprehensive view of the slow consumer problem.

5.1.2.1 Avoidance Strategies

The best way to handle this problem is, of course, to avoid it in the first place. In large part, that means keeping packets off the wire if they are not needed by the consumer(s) or likely to be dropped en route. For example, *RTI Message Service* supports powerful mechanisms for filtering your data based on its content and/or the rate at which it arrives. Messages that do not pass these filters will never be delivered to the application; in many cases, they will not even be put on the network by the producer.

See Chapter 5, "Subscribing to Messages," in the User's Manual for more information about filters and how to configure them.

5.1.2.2 Management Strategies

Despite the best efforts of an application's designers and implementers, pathological circumstances may cause consumers to fall behind. RTI provides applications with fine-grained control over the alternative behaviors listed above.

5.1.2.2.1 Send Cache Memory Management

Applications can configure how much memory a producer is allowed to use for its send queue initially. As the queue fills and then empties again, the producer will automatically adapt the rate at which it sends heartbeats to its consumers: the fuller the send queue, the more aggressively the producer will spur the consumers to acknowledge the data it has sent. The application can also receive notifications of these changes.

For information about managing the memory usage of a message producer, see Chapter 6, "Scalable High-Performance Applications: Message Reliability," in the User's Manual. For information on the more fine-grained reliability configuration options available, see Chapter 4: Data Connectivity.

5.1.2.2.1 Limited Reliability

RTI gives applications control over which old data can be removed from the send queue when it fills up. These windows of valid data can be defined in terms of time (the maximum "time to live" between when a message is written and when it should be consumed) and/or space (the "depth" of old messages to be stored in the "history"). See Chapter 6, "Scalable High-Performance Applications: Message Reliability," in the User's Manual for more information about these features.

If this level of reliability is sufficient, the message producer can be completely isolated from slow consumers by disabling positive acknowledgements. In this reliability mode, a producer informs its consumers that they only need to provide NACKs, not ACKs. Because the producer does not expect ACKs from any consumer, a slow consumer cannot affect it. See Chapter 4: Data Connectivity for more information about this configuration.

5.1.2.2.1 Consumer Inactivation

At some point, a producer can no longer maintain resources on behalf of a consumer that is not keeping up. RTI provides fine-grained control over:

- ❑ The rate at which heartbeats are sent from the producer to its consumers. See Chapter 4: Data Connectivity.
- ❑ The number of heartbeats a producer will send to a consumer without response before marking it as *inactive*. See below.

A consumer that is inactivated will not be forgotten entirely, but unacknowledged data will not be maintained solely on its behalf; communication will proceed in a best-effort-like mode with respect to that consumer. Should the consumer become responsive again, any data that it missed and that is still available for other reasons will be made available to it. For more information about this facility, see Chapter 6: Fault Tolerance.

5.2 Managing Traffic Surges

Dealing with Slow Consumers (Section 5.1.2) describes how you can deal with the situation in which message consumers cannot keep up with producers. Problems can also occur if consumers respond too promptly. If many consumers miss the same message(s), they may all NACK at once, flooding the network with reliability meta-traffic and preventing application data from flowing.

This problem can be multiplied when using multicast, since resent data will be seen by all consumers, even those that received the previous messages correctly. In the worst case, the processing and storage resources consumed by these unnecessary resends can starve out the processing of new data, leading to a self-perpetuating feedback loop of NACKs and resends ricocheting back and forth across the network.

There are three ways to reduce the damage done by surges in ACK/NACK traffic:

1. Reduce ACK/NACK volumes overall.
2. Smooth NACK spikes to avoid short-term network flooding.
3. Prevent longer-term network flooding caused by poorly targeted NACK responses.

5.2.1 Step 1: Prune and Shape Network Traffic to Reduce (N)ACKs

Some of the strategies for avoiding slow consumers can also help to prevent NACK storms. Specifically, by keeping unnecessary traffic off the network in the first place, the middleware removes the need for a consumer to ACK/NACK it, reducing the probability of a storm. These strategies are discussed in Managing Traffic Surges (Section 5.2).

5.2.2 Step 2: Wait Before Responding to Avoid NACK Storms

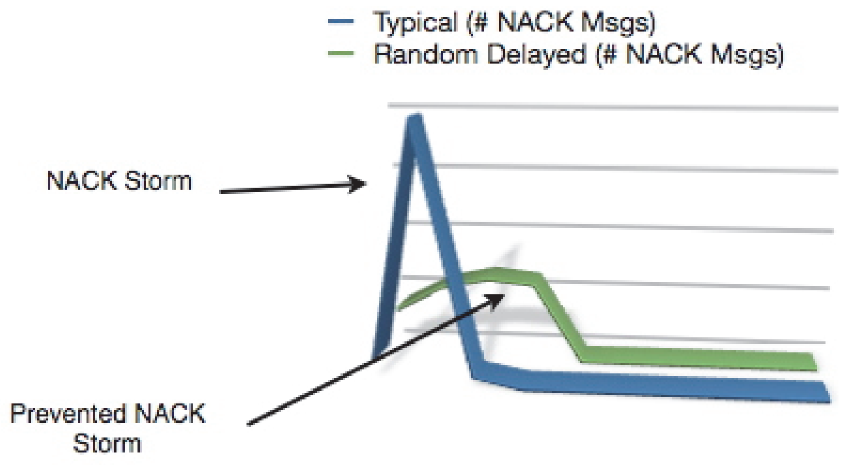
RTI provides for heartbeat and NACK "response delays": back-off times during which a producing or consuming application will refrain from putting traffic on the wire, with the expectation that others may be attempting to write at the same time.

- ❏ The "heartbeat response delay" specifies how long after receiving a heartbeat from a producer a consumer will wait before responding with an ACK or NACK.
- ❏ The "NACK response delay" governs traffic in the other direction, allowing a producer to wait before resending messages to a consumer.

These delays are specified in terms of minimum and maximum values; the actual delay will be some random value in between them. As seen in Figure 5.2, "Nack Storm Prevention With Random Delays," on page 5-10, this use of a randomly timed response, configured across a time window, causes NACKs and resent messages to be spread out in the time window instead of creating peaks of bandwidth usage.

Without a random response delay, NACKs can occur all at once, causing a spike in network traffic, as shown conceptually in the diagram above. This spike can deny network access to live application data. RTI uses random delays to smooth out those spikes, allowing data to flow normally.

Figure 5.2 **Nack Storm Prevention With Random Delays**



The following XML shows the default values of these parameters; they can be used separately or together:

```
<topic name="Example Topic">
    <producer_defaults>
        <protocol>
            <rtps_reliable_producer>
                <min_nack_response_delay>
                    <sec>0</sec>
                    <nanosec>0</nanosec>
                </min_nack_response_delay>
                <max_nack_response_delay>
                    <sec>0</sec>
                    <nanosec>200000000<!-- 200 ms --></nanosec>
                </max_nack_response_delay>
            </rtps_reliable_producer>
        </protocol>
    </producer_defaults>
    <consumer_defaults>
        <protocol>
            <rtps_reliable_consumer>
                <min_heartbeat_response_delay>
                    <sec>0</sec>
                    <nanosec>0</nanosec>
                </min_heartbeat_response_delay>
                <max_heartbeat_response_delay>
                    <sec>0</sec>
                    <nanosec>500000000<!-- 500 ms --></nanosec>
                </max_heartbeat_response_delay>
            </rtps_reliable_consumer>
        </protocol>
    </consumer_defaults>
</topic>
```

Configuring both the minimum and maximum delays to zero will cause the middleware to always respond immediately. This will make the middleware more responsive, provided that all of these responses can get through; this configuration may improve performance when the network is not heavily loaded. But if the network is congested, zero delay can lead to NACK storms.

It is also possible to configure the middleware to ignore potentially-duplicate meta-traffic altogether. For example, suppose the following sequence of events:

1. A producer sends a heartbeat to a consumer indicating which messages it has sent.
2. The consumer receives the heartbeat, realizes it has missed a message and sends a NACK.
3. In the mean time, the producer sends another heartbeat. The consumer receives it and sends another NACK, not having yet received a repair.
4. The producer receives the first NACK and sends a repair.
5. The producer receives the second NACK. *Because it cannot know whether this second NACK was sent before or after its repair would have been received, it must send another repair.*
6. The consumer receives both repairs. It delivers the first and silently discards the duplicate.

If you expect heartbeats to arrive faster than messages, you may want to avoid the extraneous message resends described above. You can do this by configuring a non-zero *NACK suppression duration* (the default is zero):

```
<topic name="Example Topic">
    <producer_defaults>
        <protocol>
            <rtps_reliable_producer>
                <nack_suppression_duration>
                    <sec>0</sec>
                    <nanosec>100000000<!-- 100 ms --></nanosec>
                </nack_suppression_duration>
            </rtps_reliable_producer>
        </protocol>
    </producer_defaults>
</topic>
```

Any duplicate NACKs that a producer receives within that duration will receive no response.

A similar parameter, the *heartbeat suppression duration*, exists on the consumer side. Unlike on the producer side, this delay is non-zero by default:

```
<topic name="Example Topic">
    <consumer_defaults>
        <protocol>
            <rtps_reliable_consumer>
                <heartbeat_suppression_duration>
                    <sec>0</sec>
                    <nanosec>62500000<!-- 62.5 ms --></nanosec>
                </heartbeat_suppression_duration>
            </rtps_reliable_consumer>
        </protocol>
    </consumer_defaults>
</topic>
```

5.2.3 Step 3: Use Multicast Intelligently to Prevent Feedback Loops

RTI Message Service can use both unicast and multicast addresses and *switch from one to the other* seamlessly and intelligently to isolate slow consumers from their better-behaving peers, helping to prevent the feedback loops of redundant resends and re-acknowledgements that can result from a surge in NACK traffic.

First, consumers can be configured to listen for messages on either unicast or multicast addresses. In topologies in which the number of consumers is limited, unicast addressing can provide superior isolation and decoupling without significantly impacting performance. In this scenario, all repair traffic will be targeted to specific consumers, avoiding increased loads on well-behaved consumers. For more information about configuring the addresses used by the middleware, see Chapter 4: Data Connectivity.

Second, even when the middleware is configured to send application messages over multicast, consumers will respond with NACKs over unicast to the specific producer whose data they are missing. The producer, in turn, can respond with message repairs either over unicast, for maximum isolation of a small number of slow consumers, or multicast, for efficiency in the case where many consumers need repairs. How it does this depends on its configured NACK response delay and the number of NACKs it receives before the delay elapses.

❑ If the producer is configured with a zero *NACK response delay* (see Step 2: Wait Before Responding to Avoid NACK Storms (Section 5.2.2)), it will respond to every NACK immediately via unicast.

- ❑ If the NACK response delay is non-zero, the producer will wait until the delay elapses before deciding what kind of addressing to use. If, by the time the delay elapses, the producer has received NACKs from multiple consumers associated with the same multicast address, the producer will send the repair to that multicast address. If all NACKs originate from unique addresses, the producer will respond over unicast to only those consumers that are not up to date.

This behavior limits the ability of poorly behaved consumers from bringing down the rest of the network in several ways:

- ❑ Consumers are decoupled from each other. Since one consumer does not depend on any other to NACK its missed data, one misbehaving consumer cannot cause another to also misbehave or lose data.
- ❑ A single slow consumer will never lead to extraneous resends to up-to-date consumers.
- ❑ The middleware can provide robustness in the face of multiple slow consumers in one or more of several ways:
    - By responding to each of them independently over unicast, so that up-to-date consumers receive no duplicate messages that they will have to discard.
    - By configuring different groups of consumers with different multicast addresses to allow multiple repairs to be sent efficiently over multicast while limiting the impact on up-to-date consumers.
    - By disabling positive ACKs (see Chapter 4: Data Connectivity) to prevent unnecessary feedback to the producer in the event that redundant resends do occur.

Chapter 6 Fault Tolerance

The User's Manual describes how to use the JMS APIs and RTI configuration mechanisms to create messaging applications. The advanced chapters of that manual, and the earlier chapters of this manual, went a step further by teaching you, in-depth, about the reliability, durability, and discovery mechanisms that help you build a robust and scalable system. But when that system is mission- and/or life-critical, that's not enough. You need to assume that something will eventually go wrong, and when it does, you need to be notified of that problem and you need the tools to respond.

This chapter will take you step-by-step through the fault tolerance mechanisms in the RTI middleware, from receiving notifications when messages don't arrive on time to automatically failing over from one publisher to another. It includes the following sections:

6.1 Data Determinism: Enforcing Periodic Deadlines

This section applies to applications with periodic or semi-periodic message flows. If your application sends messages only sporadically, you can skip to the next section.

Many applications rely on the regular arrival of messages. *RTI Message Service* can enforce this periodicity contract on behalf of your application, giving you notifications if the declared contract is broken.

This feature has two parts, each enforced separately on message producers and consumers:

- ❑ Endpoints declare their *deadline* contracts in the configuration file.
- ❑ Message producers *offer* a deadline, promising to publish a message at least once each deadline period.
- ❑ Message consumers *request* a deadline, during which they expect to receive at least one message from any producer that is publishing to them.
- ❑ The middleware issues status notifications whenever a deadline is violated, so that the application can respond appropriately.
- ❑ A *StatusNotifier* will issue an *offered deadline missed* notification if the offered deadline of a *MessageProducer* of its *Session* elapses without that producer having sent a message.
- ❑ A *StatusNotifier* will issue a *requested deadline missed* notification if the requested deadline of a *MessageConsumer* of its *Session* elapses without that consumer having received a message.

The middleware will ensure that the offered and requested deadlines of a producer/consumer pair are compatible before it will allow communication to proceed. Specifically, the deadline period offered by the producer must be shorter than or equal to that requested by the consumer. If such is not the case, an incompatible QoS notification will be provided; see below.

6.1.1 Incompatible QoS Notifications

When producers and consumers cannot communicate because of mismatched QoS configurations, your application will be notified.

Offered incompatible QoS notifications, described in Table 6.1 on page 6-3, pertain to producers.

Requested incompatible QoS notifications, described in Table 6.2 on page 6-3, pertain to consumers.

Table 6.1 **Notification Type: StatusNotifier. OFFERED_INCOMPATIBLE_QOS_NOTIFICATION_TYPE**

A producer in this session has the same Topic as a consumer, but the two have incompatible QoS policies-such as the deadline policy.

Attribute Name	Attribute Type	Description
totalCount	int	The total number of times that the producer has discovered an otherwise-matching consumer with incompatible QoS since the producer was created.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be take any integer value.

Table 6.2 **Notification Type: StatusNotifier. REQUESTED_INCOMPATIBLE_QOS_NOTIFICATION_TYPE**

A consumer in this session has the same Topic as a producer, but the two have incompatible QoS policies-such as the deadline policy.

Attribute Name	Attribute Type	Description
totalCount	int	The total number of times that the consumer has discovered an otherwise-matching producer with incompatible QoS since the consumer was created.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may be take any integer value.

6.1.2 Declaring Deadline Contracts

Deadline periods can be declared either identically for both producers and consumers of the same topic or it can be defined separately. The following XML examples show the default values.

Example: Topic-level configuration

The following configuration will be picked up by all producers and consumers of the topic. An infinite deadline indicates that no deadline is enforced.

```
<topic name="Example Topic">
    <deadline>
         <period>
              <sec>DURATION_INFINITE_SEC</sec>
              <nanosec>DURATION_INFINITE_NANOSEC</nanosec>
         </period>
    </deadline>
</topic>
```

Example: Independent producer, consumer configuration

The example above is equivalent to the following independent producer and consumer configurations:

```
<topic name="Example Topic">
    <producer_defaults>
         <deadline>
              <period>
                   <sec>DURATION_INFINITE_SEC</sec>
                   <nanosec>DURATION_INFINITE_NANOSEC</nanosec>
              </period>
         </deadline>
    </producer_defaults>
    <consumer_defaults>
         <deadline>
              <period>
                   <sec>DURATION_INFINITE_SEC</sec>
                   <nanosec>DURATION_INFINITE_NANOSEC</nanosec>
              </period>
         </deadline>
    </consumer_defaults>
</topic>
```

Take care not to define your tolerances too tightly. For example, suppose your application calls **MessageProducer.send()** periodically based on a one-second timer. You may be tempted to define your deadline period to be one second as well. However, it is likely that non-determinism in your operating system and network and small amounts of clock skew across the nodes on your network will lead to your frequently “missing” your deadlines by small amounts. These spurious deadline misses could obscure real problems in your system.

RTI therefore recommends that you leave yourself some slack when defining your deadline. How much slack is appropriate will depend on how deterministic your operating system and network are.

❑ A real-time embedded operating system such as Wind River VxWorks, LynxWorks LynxOS, or Green Hills INTEGRITY will provide more deterministic behavior than a desktop- or server-class operating system. Of the mainstream operating systems, in RTI’s experience, Linux is typically more deterministic than Microsoft Windows. Additionally, real-time Linux distributions attempt to make the determinism of Linux approach that of real-time embedded operating systems.

❑ The determinism of an Ethernet network decreases as its load increases. The quality of your NICs, switches, and drivers also has a large impact.

If the loads on your producers and consumers differ, or if your nodes are more deterministic than the network between them, you may want to consider configuring the deadlines separately for your producers and consumers to introduce more slack.

Example: Introducing slack from producer to consumer

```
<topic name="Example Topic">
    <producer_defaults>
         <deadline>
              <period>
                   <!-- 1.0 second: -->
                   <sec>1</sec>
                   <nanosec>0</nanosec>
              </period>
         </deadline>
    </producer_defaults>
    <consumer_defaults>
         <deadline>
              <period>
                   <!-- 1.1 seconds: -->
                   <sec>1</sec>
                   <nanosec>100000000</nanosec>
              </period>
         </deadline>
    </consumer_defaults>
</topic>
```

6.1.2.1 Deadlines and Keys

This section applies only to topics that have been configured for keyed behavior. If you do not use this capability, you can skip this section. For more information about keys, see Chapter 8, "Scalable High-Performance Applications: Keys," in the User's Manual.

If your topic is keyed, any deadline applies to *all instances*. That is, to offer a deadline on a keyed topic is to commit to sending a message for each of your key values at least once every deadline period.

For example, suppose that your application distributes stock information once per second, and you have used stock symbols as your keys. Once you have sent a message with a particular key value (say, "AAPL"), you have committed to continue sending messages with that key value according to your declared deadline.

6.1.2.2 Deadlines and Time-Based Filters

This section applies to time-based filters, a data sub-sampling mechanism designed to decrease network traffic and help relatively slow consumers keep up with their producer(s). If you are not using this capability, you can skip this section. For more information about time-based filters, see the Chapter 5, "Subscribing to Messages," in the User's Manual.

The Deadline QoS policy must be set consistently with the Time-Based Filter policy. For these two policies to be consistent, the deadline **period** must be longer than or equal to the **minimum_separation**. That is, you are not permitted to set a deadline so short that every message would be discarded by the time-based filter. You will not be able to create a producer or consumer that violates this rule.

For a *MessageConsumer*, the deadline and time-based filter may interact such that even though the *MessageProducer* is writing messages fast enough to fulfill its commitment to its own deadline, the *MessageConsumer* may see violations of its deadline. This happens because *RTI Message Service* will drop any messages received within the **minimum_separation**. To avoid triggering the *MessageConsumer*'s deadline, even though the matched *MessageProducer* is meeting its own deadline, set the two QoS parameters so that:

```
MessageConsumer deadline period >=
    MessageConsumer minimum_separation +
    MessageProducer deadline period
```

Example

```
<topic name="Example Topic">
    <producer_defaults>
        <deadline>
            <period>
                <!-- 1 sec: -->
                <sec>1</sec>
                <nanosec>0</nanosec>
            </period>
        </deadline>
    </producer_defaults>
    <consumer_defaults>
        <time_based_filter>
            <minimum_separation>
                <!-- 0.5 sec: -->
                <sec>0</sec>
                <nanosec>500000000</nanosec>
            </minimum_separation>
        </time_based_filter>
        <deadline>
            <period>
                <!-- 1.5 sec: -->
                <sec>1</sec>
                <nanosec>500000000</nanosec>
            </period>
        </deadline>
    </consumer_defaults>
</topic>
```

6.1.3 Missed Deadline Notifications

When a message producer fails to publish a message within its offered deadline period, any *StatusNotifier* attached to that producer's *Session* will receive an *offered deadline missed* notification. This is the case regardless of the requested deadline(s) of any matched message consumers. See Table 6.3, "Notification Type: StatusNotifier. OFFERED_DEADLINE_MISSED_NOTIFICATION_TYPE," on page 6-8.

When a message consumer fails to receive a message within its requested deadline period, any *StatusNotifier* attached to that consumer's *Session* will receive a *requested deadline missed* notification. This is the case regardless of the offered deadline(s) of any matched message producers. See Table 6.3, "Notification Type: StatusNotifier. OFFERED_DEADLINE_MISSED_NOTIFICATION_TYPE," on page 6-8.

Table 6.3 **Notification Type: StatusNotifier. OFFERED_DEADLINE_MISSED_NOTIFICATION_TYPE**

A message producer in this session has failed to publish a message within its offered deadline period.

Attribute Name	Attribute Type	Description
totalCount	int	The total number of times that the producer has failed to meet its deadline.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may take any integer value.

Table 6.4 **Notification Type: StatusNotifier. StatusNotifier. REQUESTED_DEADLINE_MISSED_NOTIFICATION_TYPE**

A message consumer in this session has failed to receive a message within its requested deadline period.

Attribute Name	Attribute Type	Description
totalCount	int	The total number of times that the consumer has failed to receive a message in accordance with its deadline.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may take any integer value.

6.2 Monitoring Liveliness and Activity

In a distributed application, a producer may need to know when a consumer becomes unresponsive, and a consumer may need to know when a producer fails. These are different but related concepts, both of which are covered in this section:

- ❑ **MessageProducer Liveliness**. *Liveliness* is a *MessageProducer*'s ability to continue publishing messages. A *MessageConsumer* can declare the rigor with which its producer must *assert*, or prove, its liveliness, and the middleware will enforce this contract. Both producers and consumers can be notified when liveliness contracts are violated.
- ❑ **MessageConsumer Activity**. *Activity* is a *MessageConsumer*'s ability to remain responsive to its producers' heartbeats. A producer defines how fast it expects its consumers to respond to these heartbeats, and if a consumer fails to respond within that time, the producer will fail the consumer over into a best-effort-like communication mode to prevent it from impacting the performance of the rest of the system. The application will also receive a notification.

6.2.1 MessageProducer Liveliness

As long as a *Connection* is up and running, the middleware will automatically *assert* the liveliness of the message producers of that connection at a frequency that can be specified by the application. The duration between liveliness assertions is referred to as the *liveliness lease duration*. When this duration elapses without a liveliness assertion, on either publishing or subscribing side, the application will receive a notification.

By default, the liveliness lease duration is infinite, meaning that the middleware need not send liveliness assertions on the network, and all message producers will be considered alive until they are closed.

```
<topic name="Example Topic">
    <liveliness>
         <lease_duration>
              <sec>DURATION_INFINITE_SEC</sec>
              <nanosec>DURATION_INFINITE_NANOSEC</nanosec>
         </lease_duration>
    </liveliness>
</topic>
```

This configuration minimizes the bandwidth the middleware uses for meta-data.

To detect whether your publishing application has crashed, hung, or been suspended, you can configure a finite liveliness lease duration. As with deadlines (see Declaring Deadline Contracts (Section 6.1.2)), you can choose to either create a single configuration to be shared by producers and consumers, or you can specify the contract separately for producers and consumers; the same compatibility rules and caveats apply.

When a message producer fails to uphold its configured liveliness contract, any *StatusNotifier* attached to that producer's *Session* will receive a *liveliness list* notification. This notification indicates that the producer *may have* lost liveliness with one or more of its consumers, depending on whether or not they are configured with the same lease duration as the producer itself and how long the interruption in service lasted. See Table 6.5.

Table 6.5 **Notification Type: StatusNotifier. LIVELINESS_LOST_NOTIFICATION_TYPE**

A message producer in this session has failed to meet its liveliness contract.

Attribute Name	Attribute Type	Description
totalCount	int	The total number of times that the producer has failed to meet its liveliness contract.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may take any integer value.

When a message producer changes its liveliness—either losing or restoring liveliness—with respect to a particular consumer, any *StatusNotifier* attached to that consumer's *Session* will receive a *liveliness changed* notification. See Table 6.6 on page 6-11.

Table 6.6 **Notification Type: StatusNotifier. LIVELINESS_CHANGED_NOTIFICATION_TYPE**

A message producer has lost or gained liveliness with respect to a consumer.

Attribute Name	Attribute Type	Description
totalCount	int	The total number of times that the producer has failed to meet its liveliness contract.
totalCountChange	int	The change to the **totalCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1. If your application polls for status changes, it may take any integer value.
notAliveCount	int	The total number of matched message producers that are currently not alive.
notAliveCountChange	int	The change to the **notAliveCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1 or -1. If your application polls for status changes, it may take any integer value.

6.2.2 MessageConsumer Activity

Unlike MessageProducer liveliness, MessageConsumer activity is a concept that applies only to reliable consumers. Producers do not maintain activity state for best-effort consumers. If you have configured your consumer for best-effort delivery, you can skip this section.

If a consumer is not keeping up with its producer(s), it can—depending on the reliability settings—impact the ability of the producer to send new messages, thereby throttling the whole system. At some point, a producer can no longer maintain unacknowledged messages on behalf of a consumer that is not keeping up if it wishes to maintain overall performance. It will internally consider the offending consumer(s) to be *inactive*, at which point it will notify the application and stop maintaining resources on behalf of those consumers.

A consumer that is inactivated will not be forgotten entirely, but unacknowledged data will not be maintained solely on its behalf; communication will proceed in a best-effort-like mode with respect to that consumer. Should the consumer become active again, any messages that it missed and that is still available will be provided to it.

This behavior is governed by the *max heartbeat retries* configuration parameter, which indicates the maximum number of periodic heartbeats that the producer will send, without receiving a response, before it will consider the consumer to be inactive. The following XML shows the default value:

```
<topic name="Example Topic">
    <producer_defaults>
         <protocol>
              <rtps_reliable_producer>
                   <max_heartbeat_retries>10</max_heartbeat_retries>
              </rtps_reliable_producer>
         </protocol>
    </producer_defaults>
</topic>
```

Figure 6.1 on page 6-13 depicts the behavior, should **max_heartbeat_retries** be set to 3:

If the consumer begins responding to heartbeats again, it will once again be marked *active* and reliable delivery can resume.

As changes in activation and inactivation occur, the application will be notified asynchronously by means of a callback. See Table 6.7 on page 6-14.

To provide higher data availability for consumers that fall behind and catch up again, as well as for consumers that may join the network late initially, you may want to configure some degree of durability/persistence for your messages. See Chapter 7, "Scalable High-Performance Applications: Durability and Persistence for High Availability," in the User's Manual for more information.

Figure 6.1 **Slow Consumer Inactivated to Clear Send Cache**

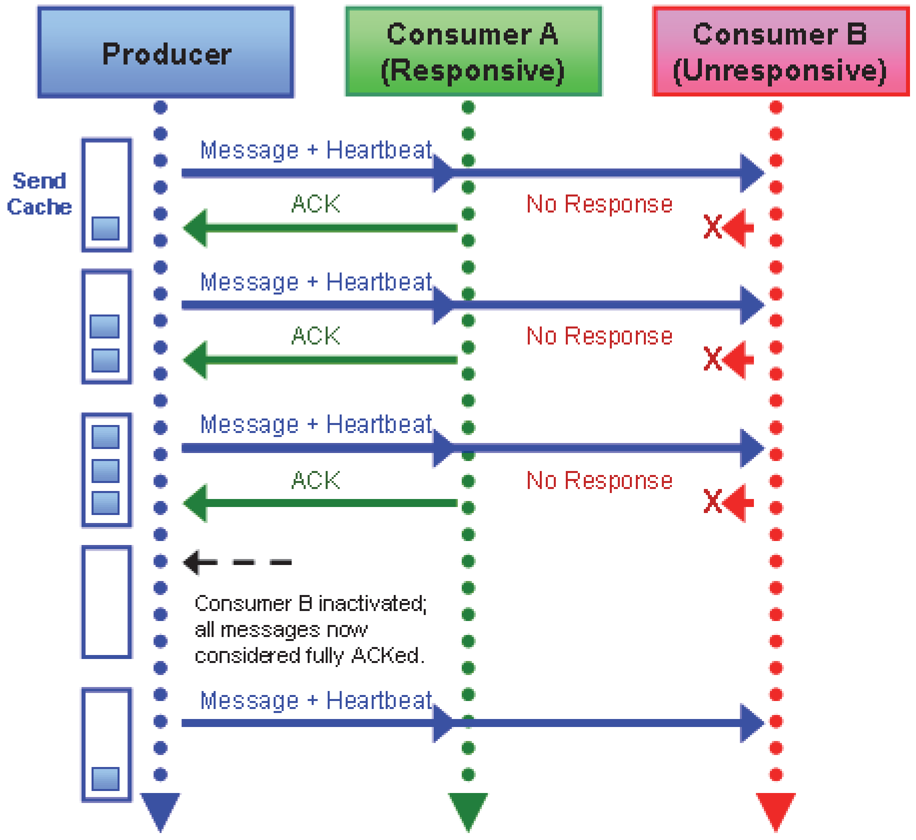


Table 6.7 **Notification Type: StatusNotifier. RELIABLE_CONSUMER_ACTIVITY_CHANGED_NOTIFICATION_TYPE**

A message producer in this session has marked a consumer inactive for failing to respond to heartbeats in a timely manner.

Attribute Name	Attribute Type	Description
activeCount	int	The total number of active reliable message consumers currently matched with this message producer.
activeCountChange	int	The change to the **activeCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1 or -1 (depending on whether the status change indicates a loss or gain of activity). If your application polls for status changes, it may take any integer value.
inactiveCount	int	The total number of inactive reliable message consumers currently matched with this message producer.
inactiveCountChange	int	The change to the **inactiveCount** attribute since the last time this status was queried. If your application receives status notifications via a listener callback, this number will generally be 1 or -1 (depending on whether the status change indicates a loss or gain of activity). If your application polls for status changes, it may take any integer value.

6.3 Ownership and Automatic MessageProducer Fail-Over

Many systems contain redundant producers of the same data; if one fails, another is supposed to take over. *RTI Message Service* can provide this functionality for your application with minimal configuration. This capability is a *hot fail-over* capability: redundant producers all publish data simultaneous; when a failure occurs, fail-over is instantaneous. The capability does not require any coordination between producers, and it does not require any consumer-to-producer back channel communication, making it low-overhead and extremely responsive.

Fail-over is based on two QoS policies that work together: *Ownership* and *Ownership Strength*. The concept is simple: if a topic is configured for *exclusive ownership*, then consumers will deliver messages only from the producer with the highest *strength*. If that producer fails—either its deadline (see Data Determinism: Enforcing Periodic Deadlines (Section 6.1)) or liveliness (see MessageProducer Liveliness (Section 6.2.1)) expires—the consumer will start delivering messages from the next-highest-strength producer automatically.

6.3.1 Configuring Ownership and Ownership Strength

There are two kinds of ownership, selected by the setting of the **kind** parameter: SHARED and EXCLUSIVE.

❏ **Shared ownership**: SHARED_OWNERSHIP_QOS indicates that *RTI Message Service* does not enforce unique ownership for the topic. In this case, message consumers will deliver messages they receive regardless of the producer from which those messages originated. *This is the default setting.*

❏ **Exclusive ownership**: EXCLUSIVE_OWNERSHIP_QOS indicates that only messages from a single producer will be delivered to the application. In other words, at any point in time, a single *MessageProducer* "owns" the message stream and is the only one whose messages will be visible to the *MessageConsumer* objects. The owner is determined by selecting the *MessageProducer* with the highest **value** of the Ownership Strength QoS policy that is currently alive, as defined by the Liveliness QoS policy, and has not violated its Deadline contract.

Ownership can change as a result of:

- ❏ A *MessageProducer* in the system with a higher strength begins sending messages.
- ❏ The *MessageProducer* that currently has ownership misses a deadline (if a finite deadline has been configured). This mechanism is appropriate for determining the ownership of topics on which messages are published periodically.
- ❏ The *MessageProducer* that currently has ownership loses liveliness (if a finite liveliness has been configured). This mechanism is appropriate for determining the ownership of topics on which messages are not published periodically.

The determination of ownership is made independently by each *MessageConsumer*. Each *MessageConsumer* may detect the change of ownership at a different time, depending on its respective configurations and the timing characteristics of the platform on which it runs. It is not a requirement that at a particular point in time all the *MessageConsumer* objects for that topic have a consistent picture of which *MessageProducer* owns the message stream.

It is possible that multiple *MessageProducer* objects with the same strength both send messages. If that occurs, *RTI Message Service* will pick one of the *MessageProducer* objects as the owner; the mechanism is internal, but all *MessageConsumer* objects will make the same choice.

Example: Configuration file, exclusive ownership for periodic data

```
<jms>
    <library name="Lib">
        <topic name="Example">
            <ownership>
                <kind>EXCLUSIVE_OWNERSHIP_QOS</kind>
            </ownership>
            <deadline>
                <period>
                    <sec>1</sec>
                    <nanosec>0</nanosec>
                </period>
            </deadline>
            <!-- Optional, to avoid having to specify ownership
              at runtime everywhere:
            <producer_defaults>
                <ownership_strength>
                    <value>10</value>
                </ownership_strength>
            </producer_defaults>
            -->
        </topic>
        <!-- Other administered objects... -->
    </library>
</jms>
```

Example: Configuration file, exclusive ownership for non-periodic data

```
<jms>
    <library name="Lib">
        <topic name="Example">
            <ownership>
                <kind>EXCLUSIVE_OWNERSHIP_QOS</kind>
            </ownership>
            <liveliness>
                <lease_duration>
                    <sec>1</sec>
                    <nanosec>0</nanosec>
                </lease_duration>
            </liveliness>
            <!-- Optional, to avoid having to specify ownership
              at runtime everywhere:
            <producer_defaults>
                <ownership_strength>
                    <value>10</value>
                </ownership_strength>
            </producer_defaults>
            -->
        </topic>
        <!-- Other administered objects... -->
    </library>
</jms>
```

Example: Application code

Assuming that all applications in the distributed system share the same version of the configuration file, setting the ownership strength will typically take place at runtime. This example shows how this could be done.

In the code for the lower-strength publishing application:

```
Hashtable<String, String> props = new Hashtable<String, String>();
props.put(
    RTIContext.QOS_FIELD_PREFIX +
    ":Lib/Example/producer_defaults/ownership_strength/value", "5");

// set other properties...
Context ctx = new InitialContext(props);
Topic myTopic = ctx.lookup("Lib/Example");

// Look up ConnectionFactory. Create Session.
MessageProducer lowStrengthPub = mySession.createMessageProducer(
    myTopic);
```

In the code for the higher-strength publishing application:

```
Hashtable<String, String> props = new Hashtable<String, String>();
props.put(
    RTIContext.QOS_FIELD_PREFIX +
    ":Lib/Example/producer_defaults/ownership_strength/value","10");

// set other properties...
Context ctx = new InitialContext(props);
Topic myTopic = ctx.lookup("Lib/Example");

// Look up ConnectionFactory. Create Session.
MessageProducer highStrengthPub = mySession.createMessageProducer(
    myTopic);
```

6.3.2 Deadlines and Keys

This section pertains to the “keys” capability of *RTI Message Service*, which allows QoS to be applied separately to different logical data objects within the same topic. SeeChapter 8, "Scalable High-Performance Applications: Keys," in the User’s Manual for more information about this capability. If you are not using keys with your topics, you can skip this section.

For keyed topics, exclusive ownership is determined on an instance-by-instance basis. That is, a subscriber can deliver messages written by a lower-strength *MessageProducer* as long as messages with that key have not been published by a higher-strength *MessageProducer*.