Progress Sonic

SonicMQ Product Family

.NET Client Guide

V7.6.2

PROGRESS SOFTWARE

Progress® SonicMQ® .NET Client Guide V7.6.2

SonicMQ and Sonic ESB Product Families include versions 8.3 and 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which is available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a "Contributor". Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a "Contributor". A small number of modules, or enhancements to modules, have been developed by other individuals (either written especially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the License (MPL-1.0.html) can also be found in the installation directory in the Docs7.6/third_party_licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

SonicMQ and Sonic ESB Product Families includes software developed by the University Corporation for Advanced Internet Development <http://www.ucaid.edu>Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development, Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

SonicMQ and Sonic ESB Product Families include XML Tools, which includes Xs3P v1.1.3. The contents of this file are subject to the DSTC Public License (DPL) Version 1.1 (the "License"); you may not use this file except in compliance with the License. A copy of the license can be found in the installation directory in the Docs7.6/third_party_licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is xs3p. The Initial Developer of the Original Code is DSTC. Portions created by DSTC are Copyright © 2001, 2002 DSTC Pty Ltd. All rights reserved.

August 2009

Contents

Preface

About This Guide

Progress SonicMQ is a fast, flexible, and scalable messaging environment that makes it easy to develop, configure, deploy, manage, and integrate distributed enterprise applications.

This book describes features of Progress SonicMQ and its .NET client.

The Progress SonicMQ features are discussed in this programming guide as follows:

- Chapter 1, “Installation and Configuration,” discusses how to install and configure the .NET Client.
- Chapter 2, “Overview,” discusses the Progress SonicMQ messaging architecture and how it it used for messaging applications. The basic concepts in this chapter provide the basis for understanding how to build efficient applications. The features in SonicMQ are summarized with references to other chapters and other books for implementation details.
- Chapter 3, “Examining the SonicMQ .NET Samples,” takes an in-depth tour through the console-based C# code samples introduced in the manual, focusing on the programming functions and features used. The VB.NET and C++ samples are also provided.
- Chapter 4, “SonicMQ Connections,” explores protocols, connection factories, connections. The identifiers and parameters of connections are presented.

  The following chapters focus on C# coding using the Progress SonicMQ .NET Client Library.
- Chapter 5, “SonicMQ Client Sessions,” explores sessions. The concepts and implementation of the transacted session and transactions are also presented. This chapter also discusses the flow control, client persistence, and integration with application servers.
- Chapter 6, “Messages,” examines the detailed composition of a message to learn what is required to construct a message, how the data populates the message, and how to manipulate messages.

- Chapter 7, “Message Producers and Consumers,” describes the scope of the session objects that produce messages and the session objects that listen, receive, and consume messages.
- Chapter 8, “Point-to-point Messaging,” explains the use of server-managed queues and discusses the similarities and differences between the Point-to-point Publish and Subscribe messaging models.
- Chapter 9, “Publish and Subscribe Messaging,” explains the characteristics unique to the broadcast type of messaging, Publish and Subscribe. Durable subscriptions, request-reply mechanisms, message selector semantics, and message listeners are presented in depth.
- Chapter 10, “Guaranteeing Messages,” describes duplicate message prevention and guaranteed message delivery. The first part of this chapter explains how you can detect duplicate messages and prevent messages from being delivered more than once. The second part of the chapter provides information about how you can use the SonicMQ Dead Message Queue (DMQ) features to guarantee that messages will not be discarded until a client has processed them.
- Chapter 11, “Hierarchical Name Spaces,” explains SonicMQ’s topic hierarchies and how they can be used to streamline access to data.
- Chapter 12, “COM Client to C# Client Migration,” describes issues to consider if you want to migrate an application written in C# using the COM client to an application that uses the C# client.

Typographical Conventions

This section describes the text-formatting conventions used in this guide and a description of notes, warnings, and important messages. This guide uses the following typographical conventions:

- **Bold typeface in this font** indicates keyboard key names (such as **Tab** or **Enter**) and the names of windows, menu commands, buttons, and other Sonic user-interface elements. For example, "From the **File** menu, choose **Open**."
- **Bold typeface in this font** emphasizes new terms when they are introduced.
- `Monospace typeface` indicates text that might appear on a computer screen other than the names of Sonic user-interface elements, including:
  - Code examples and code text that the user must enter
  - System output such as responses and error messages
  - Filenames, pathnames, and software component names, such as method names
- **`Bold monospace typeface`** emphasizes text that would otherwise appear in `monospace typeface` to emphasize some computer input or output in context.
- *`Monospace typeface in italics`* or ***`Bold monospace typeface in italics`*** (depending on context) indicates variables or placeholders for values you supply or that might vary from one case to another.

This manual uses the following syntax notation conventions:

- Brackets (`[ ]`) in syntax statements indicate parameters that are optional.
- Braces (`{ }`) indicate that one (and only one) of the enclosed items is required. A vertical bar (`|`) separates the alternative selections.
- Ellipses (. . .) indicate that you can choose one or more of the preceding items.

This guide highlights special kinds of information by shading the information area, and indicating the type of alert in the left margin.

Note A **Note** flag indicates information that complements the main text flow. Such information is especially helpful for understanding the concept or procedure being discussed.

Important An **Important** flag indicates information that must be acted upon within the given context to successfully complete the procedure or task.

Warning A **Warning** flag indicates information that can cause loss of data or other damage if ignored.

Progress SonicMQ Documentation

SonicMQ installations provide the following documentation:

- *Progress SonicMQ Installation and Upgrade Guide* — The essential guide for installing, upgrading, and updating SonicMQ on distributed systems, using the graphical, console or silent installers, and scripted responses. Describes on-site tasks such as defining additional components that use the resources of an installation, defining a backup broker, creating activation daemons and encrypting local files. Also describes the use of characters and provides local troubleshooting tips.
- *Getting Started with Progress SonicMQ* — Provides an introduction to the scope and concepts of SonicMQ messaging. Describes the features and benefits of SonicMQ messaging in terms of its adherence to the JavaSoft JMS specification and its rich extensions. Provides step by step instructions for sample programs that demonstrate JMS behaviors and usage scenarios. Concludes with a glossary of terms used throughout the SonicMQ documentation set.
- *Progress SonicMQ Configuration and Management Guide* — Describes the broker configuration toolset in detail, the certificate manager and how to use the JNDI store for administered objects. Shows how to manage and monitor deployed components including metrics and notifications.
- *Progress SonicMQ Deployment Guide* — Describes how to architect components in broker clusters, the Sonic Continuous Availability Architecture™ and Dynamic Routing Architecture®. Shows how to use the protocols and security options that make your deployment a resilient, efficient, controlled structure. Covers all the facets of HTTP Direct, a Sonic technique that enables SonicMQ brokers to send and receive pure HTTP messages.
- *Progress SonicMQ Administrative Programming Guide* — Shows how to create applications that perform management, configuration, runtime and authentication functions.
- *Progress SonicMQ Application Programming Guide*— Takes you through the Java sample applications to describe the design patterns they offer for your applications. Details each facet of the client functionality: connections, sessions, transactions, producers and consumers, destinations, messaging models, message types and message elements. Complete information is included on hierarchical namespaces, recoverable file channels and distributed transactions.
- *Progress SonicMQ Performance Tuning Guide* — Illustrates the buffers and caches that control message flow and capacities to help you understand how combinations of parameters can improve both throughput and service levels. Shows how to tune TCP under Windows and Linux for the Sonic Continuous Availability Architecture™.

- *SonicMQ API Reference* — Online JavaDoc compilation of the exposed SonicMQ Java messaging client APIs.
- *Management Application API Reference* — Online JavaDoc compilation of the exposed SonicMQ management configuration and runtime APIs.
- *Metrics and Notifications API Reference* — Online JavaDoc of the exposed SonicMQ management monitoring APIs.
- *Progress Sonic Event Monitor User's Guide* — Packaged with the SonicMQ installer, this guide describes Sonic Software's logging framework to track, record or redirect metrics and notifications that monitor and manage applications.

Other Documentation in the SonicMQ Product Family

The Progress Sonic download site provides access to additional client, and JCA adapter products and documentation:

- *Progress SonicMQ .NET Client Guide* — Packaged with the SonicMQ .NET client download, this guide takes you through the C# sample applications and describes the design patterns they offer for your applications. Details each facet of the client functionality: connections, sessions, transactions, producers and consumers, destinations, messaging models, message types and message elements. Includes complete information on hierarchical namespaces and distributed transactions. The package also includes online API reference for the Sonic .NET client libraries, and samples for C++ and VB.NET.
- *Progress SonicMQ C Client Guide* — Packaged with the SonicMQ C/C++/COM client download, this guide presents the C sample applications and shows how to enhance the samples, focusing on connections, sessions, messages, producers and consumers in both the point-to-point and publish/subscribe messaging models. Provides tips and techniques for C programmers and gives detailed information about using XA resources for distributed transactions. The package also includes online API reference for the SonicMQ C client.
- *Progress SonicMQ C++ Client Guide* — Packaged with the SonicMQ C/C++/COM client download, this guide presents the C++ sample applications and shows how to enhance the samples, focusing on connections, sessions, messages, producers and consumers in both the point-to-point and publish/subscribe messaging models. Provides tips and techniques for C++ programmers and gives detailed information about using XA resources for distributed transactions. The package also includes online API reference for the SonicMQ C++ client.
- *Progress SonicMQ COM Client Guide* — Packaged with the SonicMQ C/C++/COM client download for Windows, this guide presents the COM sample applications

under ASP, and Visual C++. Shows how to enhance the samples, focusing on connections, sessions, messages, producers and consumers in both the point-to-point and publish/subscribe messaging models. Provides tips and techniques for COM programmers. The package also includes online API reference for the SonicMQ COM client.

- *Progress SonicMQ Resource Adapter for JCA V7.6.2 User's Guide for WebSphere* — Packaged with this JCA adapter in a separate download, this guide describes the Sonic Resource Adapter for JCA and using it with a WebSphere application server.
- *Progress SonicMQ Resource Adapter for JCA V7.6.2 User's Guide for Weblogic* — Packaged with this JCA adapter in a separate download, this guide describes the Sonic Resource Adapter for JCA and using it with a Weblogic application server.
- *Progress SonicMQ Resource Adapter for JCA V7.6.2 User's Guide for JBoss* — Packaged with this JCA adapter in a separate download, this guide describes the Sonic Resource Adapter for JCA and using it with a JBoss application server.

Worldwide Technical Support

Progress Software's support staff can provide assistance from the resources on their Web site at `www.progress.com/sonic`. There you can access technical support for licensed Progress Sonic editions to help you resolve technical problems that you encounter when installing or using SonicMQ.

When contacting Technical Support, please provide the following information:

- The release version number and serial number of Progress SonicMQ that you are using. This information is:
  - Listed on your license addendum.
  - Displayed for a broker in its configuration properties' **Product Information** window.
  - Listed at the top of a SonicMQ Broker console window, similar to the following:

    ```
    SonicMQ Continuous Availability Edition [Serial Number nnn]
    Release nnn Build Number nnn Protocol nnn
    ```
- The platform on which you are running SonicMQ, as well as any other environment information you think might be relevant.
- The Java Virtual Machine (JVM) that the installation is using.
- Your name and, if applicable, your company name.
- E-mail address, telephone, and fax numbers for contacting you.

Chapter 1 Installation and Configuration

This chapter describes the download and set up of the Progress SonicMQ .NET client to run its sample applications, and to develop and run your own SonicMQ .NET messaging applications. It contains the following sections:

- "Installing the Progress SonicMQ .NET Client V7.6.1"
- "Configuring the SonicMQ .NET Client for Runtime"
- "Enabling the .NET Client for Development"

Note

Platforms and Compilers — Visit the Progress Sonic web site at `www.progress.com/sonic` for information about supported Windows platforms, processors, and compilers for the .NET client.

Downloading — The Progress SonicMQ .NET Client V7.6.2 software is available through electronic software download. Register at the Progress Software web site, `www.progress.com/sonic`, to qualify for access to the electronic download location. The download package for the .NET Client documentation includes this guide, the API online reference, the release notes, license, and readme. Each platform-specific package includes the client libraries, scripts, and sample applications.

Release Notes — See the *Progress SonicMQ .NET Client Release Notes* in the documentation package for information about known issues with the .NET client.

Installing the Progress SonicMQ .NET Client V7.6.2

Install the Progress SonicMQ .NET Client by extracting the contents of the download packages on an appropriate Windows system to a target directory of your choice, referred to hereafter as *net_client_install_dir*.

The directory structure in *net_client_install_dir* is as follows:

```
bin\
net_client_docs\
samples\
scripts\
net_client_documentation.htm
net_client_readme.htm
```

Configuring the SonicMQ .NET Client for Runtime

The libraries in a SonicMQ .NET client installation's `\bin` directory support the SonicMQ .NET client sample applications, and .NET applications that you create. The .NET framework runtime is required for running .NET applications on a Windows system.

You can see which .NET framework versions are installed on a Windows system by examining the subfolders in a system's **Windows\Microsoft.NET\Framework**.

Note **Older Windows Systems** — On supported Windows platforms prior to Windows 2003, the .NET framework is typically not pre-installed. For such systems, you must download and install a .NET framework from Microsoft, typically V1.1.

Registering the DLLs in the Global Assembly Cache

The Progress SonicMQ .NET Client installation supplies several DLLs in its **`\bin`** directory, and scripts in its \script directory to register (and when necessary, to unregister) these DLLs in the Global Assembly Cache (GAC). The GAC utility on most Windows platforms is **`gacutil.exe`**.

◆ To register DLLs in the Global Assembly Cache:

1. Open a command prompt on a system where you installed the SonicMQ .NET client.
2. Extend the `PATH` to include the location of `gacutil.exe`, typically for 1.1:
   ```
   set PATH=%PATH%;C:\WINDOWS\Microsoft.NET\Framework\v1.1.4322
   ```
3. Change directory to *net_client_install_dir*`\scripts`.
4. Enter **`register`**.

The `register.bat` script executes and adds the Sonic .NET libraries to the global assembly cache. The libraries added are `Sonic.Client`, `Sonic.Client.Jms.Impl`, `Sonic.Jms`, `Sonic.Jms.Cf.Impl`, `Sonic.Security`. Also added are third-party libraries in the SonicMQ .NET client package, `antlr.runtime`, and `Org.Mentalis.Security`.

You can now run the sample applications. But you must enable the system for .NET client development before you can extend samples or create SonicMQ .NET applications.

Enabling the .NET Client for Development

To develop applications with the Progress SonicMQ .NET Client, you must also install the appropriate .NET Framework SDK, available from Microsoft, typically in a Visual Studio package.

Important **Other Development Resources** — The Windows development platform, .NET Framework version, and language of your source programs might require additional third-party resources to edit, compile, deploy, and provision SonicMQ .NET applications on your target platforms. Also see the *`net_client_install_dir`*`\samples\readme.txt` file for additional information about building and running the .NET client samples.

SonicMQ .NET Client Application Programming Interfaces

The following SonicMQ .NET client DLLs expose APIs:

- **`Sonic.Jms.Cf.Impl.DLL`** — Contains the `Sonic.Jms.Cf.Impl` namespace
- **`Sonic.Jms.DLL`** — Contains the `Sonic.Jms`, `Sonic.Jms.Ext, and Sonic.XA` namespaces
- **`Sonic.Security.DLL`** — Contains the `Sonic.Security.Pass.Client` namespace

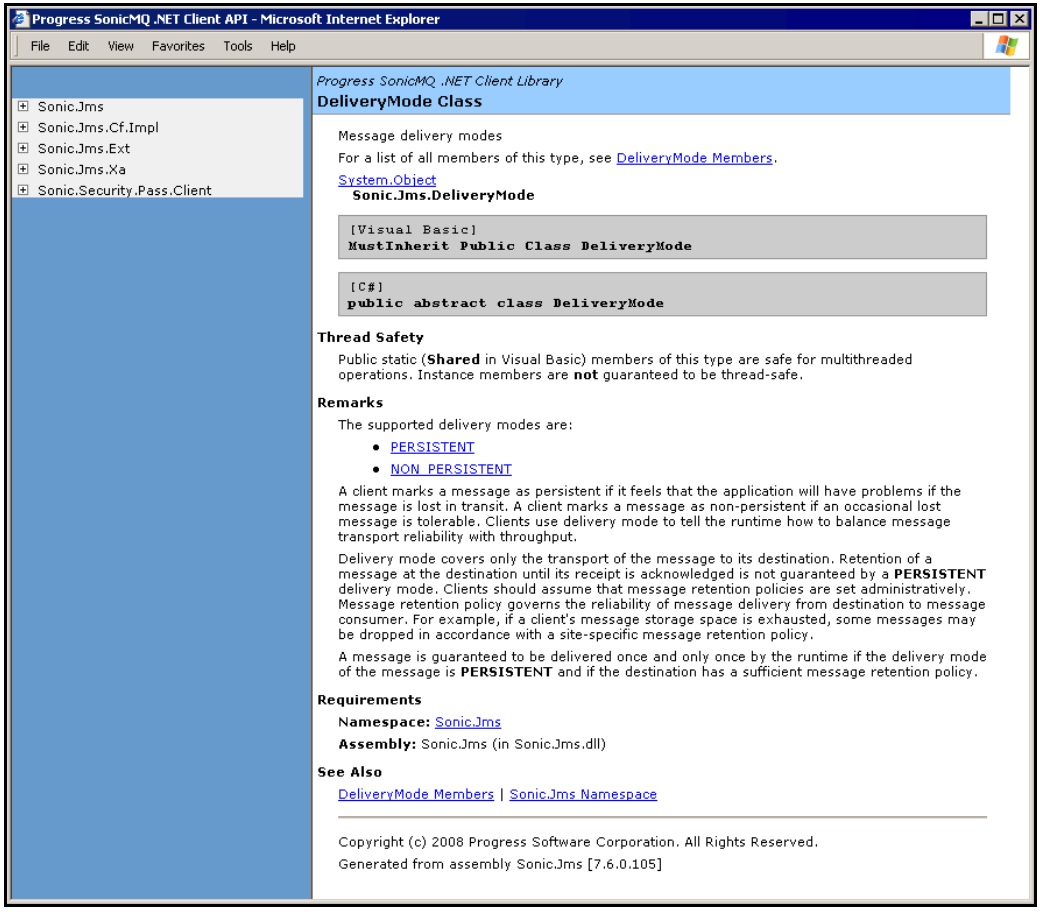
SonicMQ .NET Client Online API Reference

The Progress SonicMQ .NET Client Library API consists of the following namespaces:

- **`Sonic.Jms`** — Contains interfaces and classes used with SonicMQ
- **`Sonic.Jms.Cf.Impl`** — Contains the `ConnectionFactory` class, from which most other C# client objects are created
- **`Sonic.Jms.Ext`** — Contains additional interfaces and classes used with SonicMQ
- **`Sonic.Jms.XA`** — Contains the classes and interfaces for XA transactions
- **`Sonic.Security.Pass.Client`** — Contains interfaces for pluggable authentication

The documentation for the SonicMQ .NET Client API is accessible through the installed documentation page *net_client_install_dir*\net_client_docs\net_documentation.htm, or directly at *net_client_install_dir*\net_client_docs\api\index.html.

In the following example, the **Sonic.Jms** class **DeliveryMode** is shown:



The SonicMQ .NET Client installation is complete.

The next chapter provides an overview of SonicMQ messaging. After that, C# samples are explored to acquaint you with the SonicMQ .NET client functionality in applications. Additional samples in C++ and VB.NET are provided for alternative .NET client usage.

Chapter 2 Overview

SonicMQ is an efficient, secure, and scalable messaging system for business-to-business, networked, and internal integrated applications. SonicMQ makes it possible for organizations to efficiently (and reliably) communicate between disparate business systems over the Internet and meet their time-to-market requirements by delivering the following features:

- Internet-resilient business messaging
- High performance messaging infrastructure
- Reliable transmission of messages regardless of network, hardware, or application failure
- Messaging topologies that support complex deployments distributed across geographic and system boundaries:
  - Dynamic Routing Architecture (DRA) to publish and subscribe to remote nodes
  - Clusterwide access to global queues and durable subscriptions
  - Load-balanced subscriptions
- Centralized management environment that allows all components of the SonicMQ messaging infrastructure to be quickly and easily administered and monitored from a central location:
  - SonicMQ's JMX-based administration environment works across routing nodes
  - Manage and administer collections of brokers as a group
  - Fault tolerance is managed through local persisted configuration cache

- SonicMQ provides secure data transmission and controlled access with:
  - Pluggable cipher suites for Quality of Protection (QoP)
  - Pluggable client authentication
- Flexibility in configuring the messaging infrastructure:
  - Clients can be moved around the network without requiring any changes to the messaging application
  - Support for a variety of message types, including XML and Multipart
  - Option to establish persistence on the client message producer
  - Peer-to-peer file transfers over recoverable file channels
- Ease-of-use features make SonicMQ an environment that can be easily understood and deployed

Programming Concepts

SonicMQ messaging involves the loose coupling of applications. This is accomplished by maintaining an intelligent broker structure. A client can establish one or more connections to a broker.

Clients Connect to the SonicMQ Broker

SonicMQ's hub-and-spoke architecture considers every entity in the messaging service topology to be a client except the broker—the entity to which every client connects and through which all clients exchange messages.

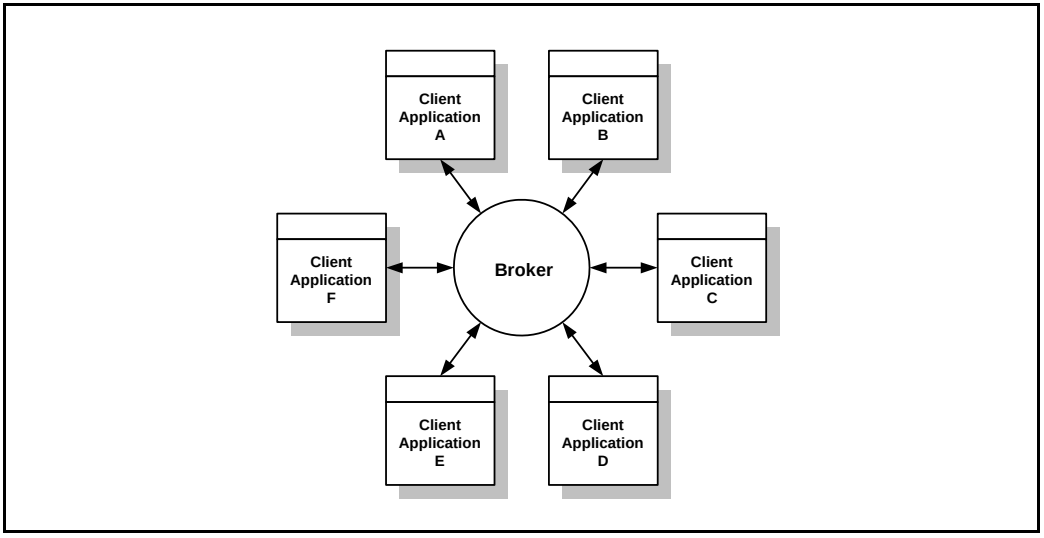


Figure 1. Broker Is a Hub for SonicMQ Client Applications

The broker can join with other brokers to form **clusters**. Clusters and stand-alone brokers are nearly equivalent when looked at as **routing nodes**.

SonicMQ .NET Client API

Figure 2 shows the .NET client API and the SonicMQ client run time within a client application. The .NETclient API provides a set of interfaces you can use to build client applications. The client run time implements the interfaces in the API.

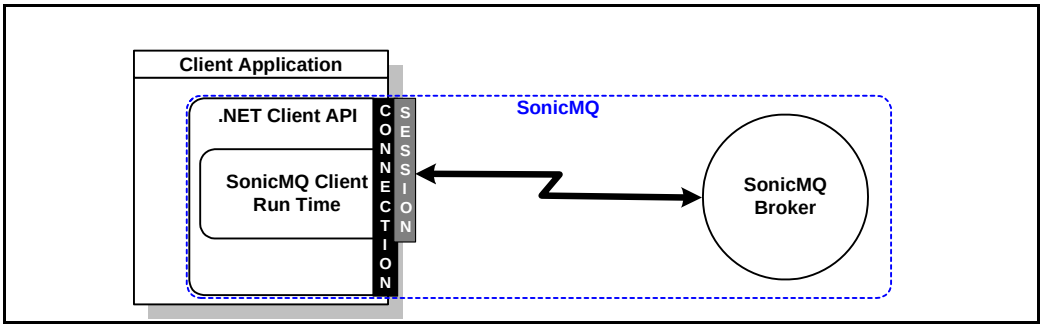


Figure 2. Client Application Using the .NET Client API

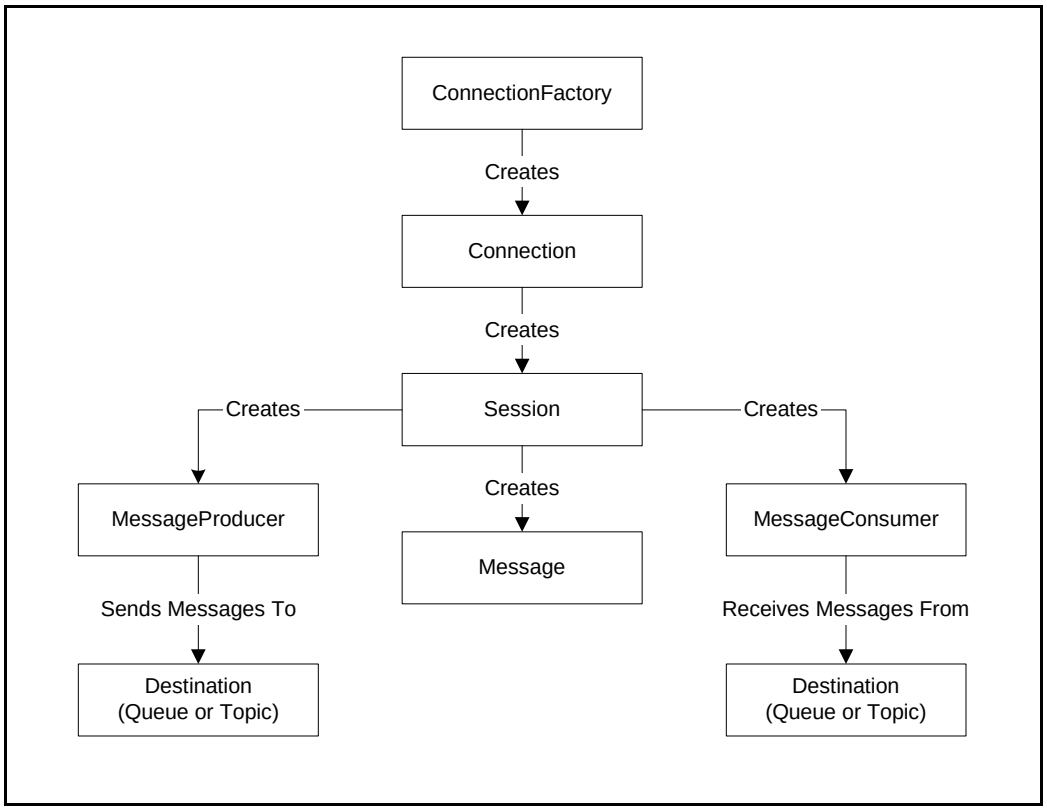
SonicMQ Messaging Models

SonicMQ supports two messaging models:

- **Point-to-point (PTP) model** — A message producer sends a message to a **queue**—a type of named destination (topics are the other type of destination). Once a message is in the queue, the message is available to multiple consumers, but only one consumer can actually accept the message. This model is referred to as a *one-to-one* form of communication, because each message is delivered to only one consumer.
- **Publish and Subscribe (Pub/Sub) model** — A message producer sends a message to a **topic**—another type of named destination. Once a message is published to a topic, the message is available to multiple consumers, who can all accept the message. This model is referred to as a *one-to-many* or *broadcast* form of communication, because each message is delivered to all (usually multiple) consumers.

SonicMQ Object Model

The following figure shows the SonicMQ object model.



ConnectionFactory

A `ConnectionFactory` is an object that creates one or more `Connection` objects, each of which establishes a connection to a SonicMQ broker (or cluster).

Connection

A `Connection` is a conduit for communication between your client application and a SonicMQ broker (or cluster of brokers). Each `Connection` is a single point for all communications between the client application and the broker.

Session

A `Connection` can create one or more `Session` objects. A `Session` object is a single-threaded context for producing and consuming messages. A `Session` object can create `Message` objects, `MessageProducer` objects (which send outbound messages), and `MessageConsumer` objects (which receive inbound messages). Each `MessageProducer` and `MessageConsumer` object operates in the context of the `Session` that created it.

Transactions are scoped to `Session` objects. A transaction can include both Point-to-Point and Pub/Sub messages.

MessageConsumer and MessageProducer

A `Session` creates `MessageProducer` and `MessageConsumer` objects. A `MessageProducer` object sends messages from your client application to destinations on the broker. A `MessageConsumer` object receives messages from a destination on the broker, either synchronously (via the `receive()` method) or asynchronously (via a `MessageListener` object).

The general terms **consumer** and **producer** refer, respectively, to entities that receive and send messages. Figure 3 illustrates the roles of producers and consumers.

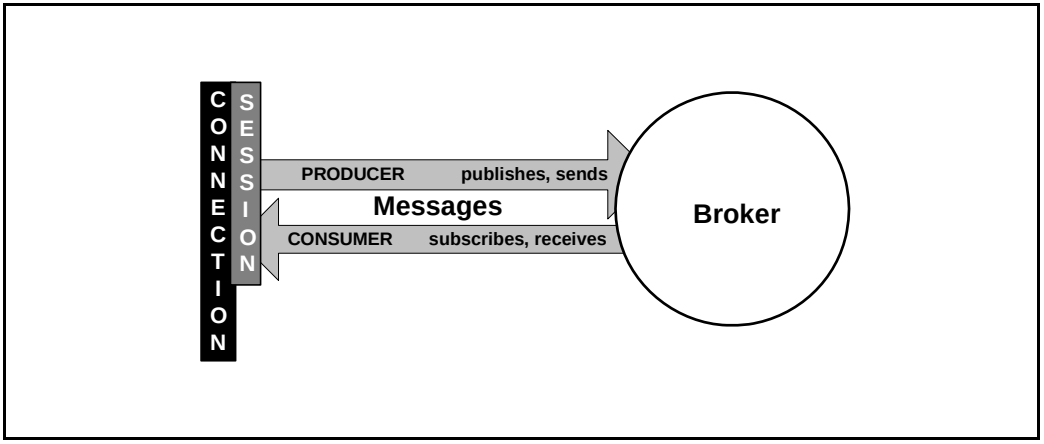


Figure 3. Message Producers and Message Consumers

The client application is the **producer** when:

- **Sending** a message to a queue (PTP)
- **Publishing** a message to a topic (Pub/Sub)

The client application is the **consumer** when:

- **Receiving** messages from a queue (PTP)
- **Subscribing** to a topic (Pub/Sub) where messages are published

A single `Session` object can create both `MessageProducer` and `MessageConsumer` objects.

To learn about the broker architecture and functionality, see the *Progress SonicMQ Deployment Guide*.

Destination

A `Destination` object represents a named location to which messages can be sent. A destination must be either a `Topic` or a `Queue` (both extend the `Destination` interface).

When you write an application using the .NET API interfaces, you can use the same interfaces for both messaging models, but you cannot create a "common" `Destination` object. You create either a `Topic` or a `Queue`, which you can then upcast to a `Destination`, if needed.

Message

A `Message` object holds your business data. For more information about messages, see Chapter 6, "Messages."

Quality of Service and Protection

Some messages are simple, transitory, and are broadcast to prospective recipients who might or might not be paying attention. These messages might contain information that is timely and important but not particularly confidential. An example is stock quotes. The data is public information that is considered valuable when it is disseminated promptly and verifiable when significant risk might be associated with the information it carries. Here, performance takes precedence.

Messages that represent the other extreme, where the anticipated services and protection are paramount, include bank wire transfers where encryption, security, and logging processes are an integral part of mutually assured confidence in the message. Communication that is certifiable, auditable, consistent, and fully credentialed provides the quality of service and the quality of protection that is expected. Performance is important, but not important as quality.

All the SonicMQ message services and protection are available with both the PTP and Pub/Sub messaging models.

The services and protection that are described in this guide—together with some of the services controlled by the broker's administrator—are shown in Table 1.

Table 1. Services and Protection Available With SonicMQ Messaging

Service	*Technique*	*Process*	*Reference*
ENCRYPTED Content is encrypted.	Independent encryption mechanisms.	Body is appended after it is encrypted, providing assurance that a message is protected even if the connection is not secure.	Private encryption methods can be applied before the message is presented to the messaging-enabled application.
SECURE TRANSPORT Protocol is secure.	Connection protocol parameter.	Parameter is set when creating connection.	See Chapter 4, "SonicMQ Connections," for information about choosing protocols.
AUTHENTIC PRODUCER Producer is accepted by the broker's authentication domain.	Security enforced through authentication of user name and password at time of connection.	If the installation enabled security, the administrator sets up users and passwords in the broker's authentication domain.	See the *Progress SonicMQ Deployment Guide* for information about authentication and authorization of producers (PTP senders and Pub/Sub publishers) and Access Control Lists (ACLs).
AUTHORIZED PRODUCER Producer has permission to produce and is authorized to produce to specified destination.	Security enforced through Access Control Lists (ACLs).	If the installation enabled security, the administrator sets up permissions in the broker's authorization policy to produce to specific hierarchies of destinations and routings.	
ACKNOWLEDGED PRODUCER Broker acknowledges receipt of messages from producer.	Synchronous block released after receipt at broker.	Automatic when sending a message unless specifically designated as in `NON_PERSISTENT_ASYNC` acknowledgement mode	

Table 1. Services and Protection Available With SonicMQ Messaging (*continued*)

Service	*Technique*	*Process*	*Reference*
INTEGRITY When a destination has a QoP setting that indicates `integrity`, a message producer creates a digest that the broker confirms. The broker recreates the message digest for the message consumer who then confirms it.	The message content is hashed and the digest of the result accompanies the message.	Administrative Quality of Protection (QoP) setting on destination is `integrity`. Also implicitly set when a sender chooses per-message encryption.	See the *Progress SonicMQ Configuration and Management Guide* and the *Progress SonicMQ Deployment Guide* for information about administrator settings for privacy and integrity. Note also information about how the installed cipher suites for QoP encryption can be customized. The `privacy` setting can be explicitly requested by a message producer to a security-enabled broker. See "Per Message Encryption" on page 158.
PRIVACY When a destination has a QoP setting that indicates `privacy`, a message producer encrypts a message then creates its digest. The broker confirms the digest and decrypts the message. The broker reencrypts the message and then recreates the message digest for the message consumer.	The message is encrypted with the cipher suite preferred by the broker and then the message content is hashed and the digest of the result accompanies the encrypted message	Administrative Quality of Protection (QoP) setting on destination is `privacy` or message producer explicitly requests `privacy`. Setting `privacy` includes the services of `integrity`.	
PERSISTENT Message persists in broker storage.	Delivery mode uses the `PERSISTENT` option.	Set option in publish or send command. The broker never allows messages to be lost in the event of a network or system failure. Nonpersistent messages are volatile in the event of a broker failure.	

Table 1. Services and Protection Available With SonicMQ Messaging (*continued*)

Service	*Technique*	*Process*	*Reference*
REDELIVERY Consumer might receive unacknowledged message again.	Broker sets `JMSRedelivered` field to `true` when service is interrupted while waiting for a consumer acknowledgement.	Must be checked and acted on by the consumer. For the message producer, this header field has no meaning and is left unassigned by the sending method.	See "Recover" on page 122.
DURABLE INTEREST Pub/Sub consumers, subscribers, can establish a durable interest in a topic with a broker.	An application uses the session method `create-DurableSubscriber` with the parameters *`topic`*, *`subscriptionName`*, *`messageSelector`*, and a *`noLocal`* option.	Broker retains messages for durable subscriber, using the *`userName`*, and *`clientID`* of the connection plus the *`subscriptionName`* to index the subscription. Note that `NON_PERSISTENT` messages are still at risk in the event of broker failure. Note also that messages expire normally even if durable subscriptions are unfulfilled.	See "Reliable, Persistent, and Durable Messaging Samples" on page 54. See also "Durable Subscriptions" on page 224.
PRIORITY Messages sent with higher priority can be expedited.	Producer sets the message header value `JMSPriority` to an `int` value `0` through `9` where `4` is the default.	Broker checks message priority and handles appropriately. Priority values of `5` through `9` are expedited.	See "Message Management by the Broker" on page 175.
EXPIRATION Messages are available until the expiration time. Based on GMT.	Producer sets `time-to-live` value, then includes the value at moment of publish/send.	Broker receives message with `JMSExpiration` date-time set to the `JMSTimestamp` date-time plus the `time-to-live` value.	See "Create the Message Type and Set Its Body" on page 171. See also "Message Management by the Broker" on page 175.

Table 1. Services and Protection Available With SonicMQ Messaging (*continued*)

Service	*Technique*	*Process*	*Reference*
REQUEST MECHANISM Producer can request a reply from the consumer.	Message header field `JMSReplyTo` has a string value that indicates the topic where a reply is expected. The `JMSCorrelationID` can indicate a reference string whose uniqueness is managed by the producer.	Carried through to consumer, but the consumer application must be coded to look at the `JMSReplyTo` field and then act. Producer could be synchronously blocked waiting for reply message at temporary topic. `TopicRequestor` object creates a temporary topic for the reply.	See "Request and Reply Samples" on page 61. See also "Session Objects" on page 125 and "Reply-to Mechanisms" on page 188.
AUTHENTIC CONSUMER Consumer is accepted by the broker's authentication domain.	Security enforced through authentication of username and password at time of connection.	If the installation enabled security, the administrator sets up users and passwords in the broker's authentication domain.	See the *Progress SonicMQ Deployment Guide* for more about authentication and authorization of consumers (PTP receivers and Pub/Sub subscribers) and Access Control Lists (ACLs).
AUTHORIZED CONSUMER Consumer is authorized to consume from a specified destination.	Security enforced through ACLs.	If the installation enabled security, the administrator sets up permissions in the broker's authorization policy to consume from specific hierarchies of destinations.	

Table 1. Services and Protection Available With SonicMQ Messaging (*continued*)

Service	*Technique*	*Process*	*Reference*
ACKNOWLEDGED CONSUMPTION Consumer acknowledges receipt to broker.	Acknowledgement type for the session was set when the session was created.	Functions automatically to perform the specified type of acknowledgement for all messages consumed in that session.	See "Acknowledgement Mode" on page 121.
Client acknowledges receipt of received messages when session parameter is `CLIENT_ACKNOWLEDGE` or `SINGLE_MESSAGE_ACKNOWLEDGE` then when client calls **`acknowledge()`**.	Explicit call by consumer.	Manual.	
REPLY MECHANISM Consumer replies to the producer's request for reply.	Consumer reacts to a `JMSReplyTo` request by producing a message to the topic name in the `JMSReplyTo` field.	Programmatic procedure where the consumer publishes a reply. The content of the reply is not specified. Typically the `JMSCorrelationID` is replicated.	See "Request and Reply Samples" on page 61. See also "Session Objects" on page 125 and "Reply-to Mechanisms" on page 188.
DEAD MESSAGE QUEUE Sender/publisher can set properties to re-enqueue undelivered messages and/or send an administrative notice.	Set the properties that tell the broker to provide special handling when the message is declared dead.	Programmatic procedure where the sender chooses to set the property `JMS_SonicMQ_preserveUndelivered` to `true` to store the dead message until handled and to set the property `JMS_SonicMQ_notifyUndelivered` to `true` to send a notification to the broker's administrator.	See "Message Properties" on page 156. See also the Dynamic Routing information in the *Progress SonicMQ Deployment Guide*.

Clients

The techniques and interfaces described in this book describe the methods and design patterns for running SonicMQ in a console session using the .NET client. There are several other client types that provide client functionality.

- **Java Client** — SonicMQ clients are a set of Java archives that provide libraries of functionality that enable applets, bridges, proxy servers, servlet engines, and JavaBeans.
- **HTTP Direct Protocol Handlers** — HTTP Direct is a broker-based set of properties and factories that enables seamless interfacing between the JMS message and HTTP document paradigms. Pure HTTP documents arriving inbound on SonicMQ broker ports are transformed into JMS messages for message production to the port's assigned destination. Outbound messages to specified SonicMQ routing nodes are transformed to HTTP documents and then sent to the designated HTTP Web Server. HTTP Direct also has features to handle SOAP encoding and to read properties from specified HTTP fields. See the *Progress SonicMQ Deployment Guide* for more information and for samples of HTTP Direct.
- **C and C++ Client** — SonicMQ can act as a pure C++ or pure ANSI C application on your system and still interface with a SonicMQ broker with the same behaviors as a pure JMS client. This provides legacy systems with integration and Web connection capabilities within the familiar operating characteristics of C and C++.
- **COM Client** — SonicMQ can act as a pure COM application on your system and still interface with a SonicMQ broker with the same behaviors as a pure JMS client. This provides legacy systems with integration and Web connection capabilities within the familiar operating characteristics of COM.

Chapter 3 Examining the SonicMQ .NET Samples

This chapter explains how to run the sample applications included with SonicMQ. These samples illustrate some of the messaging functionality of SonicMQ. This chapter includes the following sections:

About SonicMQ .NET Client Samples

Samples are provided with SonicMQ and introduced in the *Getting Started with Progress SonicMQ* guide. Those Java samples are explored in greater depth in the *Progress SonicMQ Application Programming Guide*. The Progress SonicMQ .NET Client provides substantially the same samples in C#, as well as a few of those samples in C++ and VB.NET. In this chapter, the functionality of the samples is explored in more detail to illustrate some of the features of SonicMQ.

When you run the samples, the standard input and standard output displayed in the console can represent data flows to and from a range of applications and Internet-enabled devices such as:

- **Application software** for accounting, auditing, reservations, online ordering, credit verification, medical records, and supply chains
- **Information appliances** such as beepers, cell phones, wireless devices, fax machines, and Personal Digital Assistants (PDAs)
- **Real-time devices** with embedded controls such as monitor cameras, medical delivery systems, climate control systems, and machinery
- **Distributed knowledge bases** such as collaborative designs, service histories, medical histories, and workflow monitors

Important

When you install SonicMQ with the default options, security is disabled by default, and the SonicMQ broker is configured to listen for connections on port 2506 (on the local machine). The samples in this chapter—and the instruction to run them—assume that you are using the defaults after a typical installation: they assume security is disabled and the broker is listening on port 2506. However, if needed, you can run these samples against another broker listening on a different port. To do this, when you run a sample, add the "-b" option to the command line and specify the hostname and port (for example, `-b otherhost:2508`).

Because the samples assume that security is disabled by default, any user names and passwords provided on the command line to run the samples are treated as arbitrary strings; if security is enabled, all user names and passwords must be configured beforehand. See the *Progress SonicMQ Deployment Guide* for information about how to implement a SonicMQ sample in a secure environment.

C# Samples

The SonicMQ .NET client provide extensive C# samples to demonstrate the following basic features of SonicMQ:

- **Chat and Talk Samples** — The basic messaging functions are demonstrated by producing and consuming messages using both messaging models (PTP and Pub/Sub):
  - **`Talk`** (PTP)
  - **`Chat`** (Pub/Sub)
- **Transaction Samples** — Transactions are shown in both domains in application windows to show how the producers and consumers of the transacted messages see the messages flow:
  - **`TransactedTalk`** (PTP)
  - **`TransactedChat`** (Pub/Sub)
- **Additional Message Types** — To simplify input, the preceding examples are Text messages. The following samples display other common message types in the messaging domains:
  - **MapMessages** — `MapTalk` (PTP)
  - **Decomposing MultiPart Messages** — `Multipart` (PTP)
- **Reliable, Persistent, and Durable Messaging** — These samples demonstrate techniques that can enhance the Quality of Service. Reliable connections show how to keep connections active in both domains. The durable subscription sample shows how Pub/Sub subscribers can have messages held for them.

  The samples in this category are:
  - **Reliable Connection** — `ReliableTalk` (PTP), `ReliableChat` (Pub/Sub)
  - **Durable Subscription** — `DurableChat` (Pub/Sub)
- **Request and Reply** — These transacted examples show the mechanisms for the producer requesting a reply and the consumer fulfilling that request:
  - **Originator's Request** — `Requestor` (PTP, Pub/Sub)
  - **Receiver's Response** — `Replier` (PTP, Pub/Sub)
- **Test Loop** — This sample, **Queue Round Trip (PTP),** shows how quickly messages can be sent and received in a test loop.

- **Selection and Wild Cards** — The message selector samples use SQL syntax to qualify the messages that are visible to an application while the `HierarchicalChat` sample uses template characters to subscribe to a set of topics that is qualified when messages are published:
  - **Message Selection** — `SelectorTalk` (PTP), `SelectorChat` (Pub/Sub)
  - **Wild Cards** — `HierarchicalChat` (Pub/Sub)

VB.NET and C++ Samples

The Progress SonicMQ .NET client functions in a managed .NET environment. This environment enables an application programmer to utilize functionality provided by the .NET client using the VB.NET and C++ languages, samples for which are located in the `vbdotnet` and `cplusplus` folders. See the *`net_client_install_dir`*`\samples\readme.txt` file for more information about how to configure, build, and run these samples.

Other SonicMQ Samples Available

There are several other SonicMQ samples that require special setup to explore them.These samples are described in other chapters of this book, or in other SonicMQ documents.

- **Dynamic Routing Queues** — When routing queues are established across brokers, messages are dynamic. The **`GlobalTalk`** (PTP) sample demonstrates dynamic routing queues once you have an appropriate setup. See the “Multiple Nodes and Dynamic Routing” chapter in the *Progress SonicMQ Deployment Guide* for information about this sample.
- **Replicated (High Availability) Brokers** — See the *Progress SonicMQ Deployment Guide* for an example of how you can set up brokers as a primary/backup pair. When the brokers are running and replicating, you can stop the active broker, causing the standby broker to fail over. You can run the fault tolerant example—see “Modifying the Chat Example for Fault-Tolerance” on page 110—to see the client application seamlessly continue its session on the broker that becomes active.
- **Secure Socket Layer (SSL)** — SSL samples show how to reconfigure the broker for SSL security, how to run client-side applications that connect through SSL, and how to use certificates. See Part II of the *Progress SonicMQ Deployment Guide* for complete SSL implementations you can explore. These implementations use the RSA security software and credential samples installed with SonicMQ.
- **Security Enabled Dynamic Routing** — See the *Progress SonicMQ Deployment Guide* for an example of how you can set up multiple brokers and security to realize secure dynamic routing across nodes.

- **Advanced Security** — The sample folder `AdvancedSecurity` provides the source and runtime of a sample application, and DLL that enables C# clients to use Login SPI security features in your applications.

  See the *Progress SonicMQ Administrative Programming Guide* chapter "Using the Advanced Security SPIs for the Sonic Authentication Domains" to see how to use this C# sample and the Java sample to illustrate implementation of the Pluggable Authentication and Security Services (PASS) SPIs.

  In a Progress SonicMQ installation that includes samples, see the file *`MQ7.6.2_install_dir`*`\samples\AdvancedSecurity\readme.htm` for information about samples that demonstrate advanced security features using the PASS SPIs.

  For more information about the PASS features in SonicMQ, see the chapter "Security Considerations in System Design" in the *Progress SonicMQ Deployment Guide*.

Extending the Samples

After reviewing the sample applications, you can explore some variations by changing the source files. You can edit the source files, compile the changed file, and then run the applications again to observe the effect. For example:

- Using a common topic for two samples.
- Observing how different messaging behaviors affect round-trip times.
- Modifying `MapMessage` to use other data types.
- Modifying `Chat` to use fault tolerant connections and display connection URLs.

How Security Impacts Client Activities

Security provides the high quality of protection and access by applications that is expected in enterprise applications. The section "Quality of Service and Protection" on page 30 provides an overview of the features and functions of security. However, unless the broker enables security and the broker's persistent storage mechanism is initialized for security, security is not enabled.

The samples in this chapter do not use security so that you can begin exploring the messaging features without having to first set up security objects for:

- **User authentication** — When security is activated, only defined usernames are allowed to connect to the broker.
- **User authorizations** — The administrator can control a user's ability to perform actions such as subscribing to a topic and reading from queues.

See the *Progress SonicMQ Deployment Guide* for information about how to implement a SonicMQ sample in a secure environment.

Running the SonicMQ .NET Client C# Samples

The following sections explain the tasks required to start SonicMQ to work with the C# client sample applications:

Starting the SonicMQ Container and Sonic Management Console

Before you can run the C# client samples, you must first connect to a SonicMQ domain manager. The domain manager can be installed locally or remotely. When SonicMQ is first installed (a typical installation), a default domain manager is installed (Container1). The C# client samples assume that you have installed SonicMQ locally and are using the default domain manager (Container1).

Be sure the SonicMQ container is running before executing any of the C# client samples. The following procedures explain how to start the SonicMQ container and Sonic Management Console. For more detailed information on working with the Sonic Management Console, see the *Progress SonicMQ Configuration and Management Guide*.

Note If this is the first time you are running SonicMQ, you should not have to set up and initialize the persistent storage mechanism or adjust the broker's `settings`. See the *Progress SonicMQ Installation and Upgrade Guide* for more information.

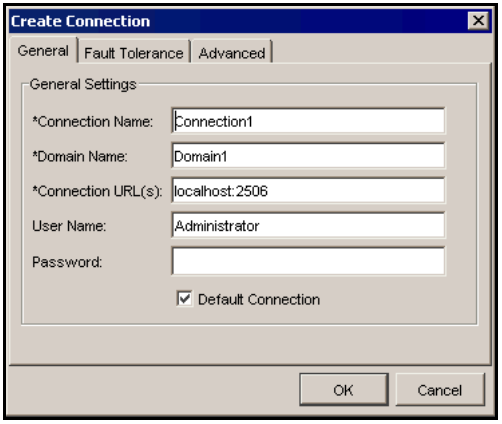
◆ To start the broker process from the Windows Start menu:

1. Select **Start > Programs > Progress > Sonic 7.6 > Start DomainManager**.

   SonicMQ starts the container that hosts the broker and then starts the broker.

2. Select **Start > Programs > Progress > Sonic 7.6 > Sonic Management Console**.

The management console opens the **Create Connection** dialog box:



3. If you did not enable security when you installed, you can accept the defaults in the **General** tab in the **Create Connection** dialog box.

   If you enabled security when you installed SonicMQ, enter your password.

Important When you install SonicMQ (a typical installation), security is disabled by default, and the SonicMQ broker is configured to listen for connections on port 2506 (on the local machine). The samples in this chapter—and the instruction to run them—assume that you are using the defaults after a typical installation: they assume security is disabled and the broker is listening on port 2506. However, if needed, you can run these samples against another broker listening on a different port. To do this, when you run a sample, add the "-b" option to the command line and specify the hostname and port (for example, `-b otherhost:2508`).

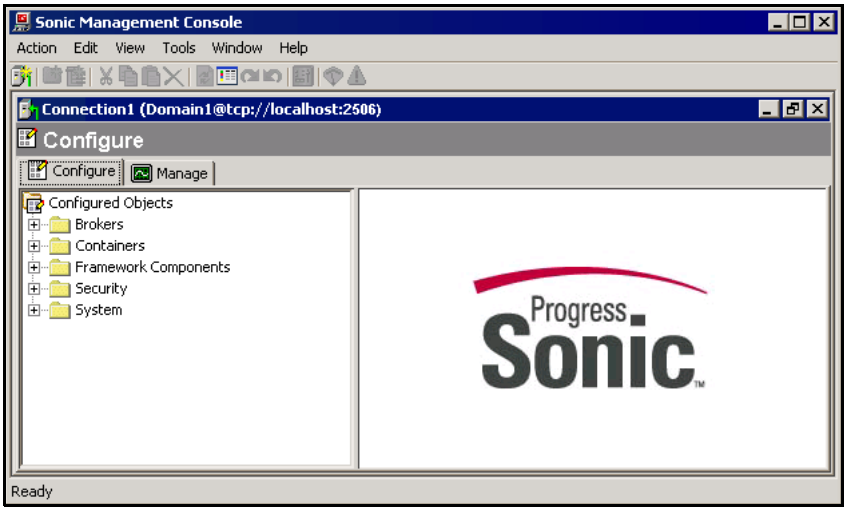
Because the samples assume that security is disabled by default, any user names and passwords provided on the command line to run the samples are treated as arbitrary strings; if security is enabled, all user names and passwords must be configured beforehand. See the *Progress SonicMQ Deployment Guide* for information about how to implement a SonicMQ sample in a secure environment.

   See the *Progress SonicMQ Configuration and Management Guide* for information about setting parameters under the **Advanced** tab. For these samples, you do not have to set any advanced parameters.

4. Click **OK**.

A **Connecting...** dialog box and the status bar indicate that the Sonic Management Console is connecting to the broker.

The Sonic Management Console opens to the **Configure** view:



See the *Progress SonicMQ Configuration and Management Guide* for information about configuration and management using the Sonic Management Console.

Opening Client Console Windows

Each application instance is intended to run in its own console window with the current path in the selected sample directory.

Chat and Talk Samples

The fundamental differences between Pub/Sub and PTP messaging are demonstrated in the `Chat` and `Talk` samples.

Chat Application (Pub/Sub)

In the `Chat` application, whenever anyone sends a text message to a given topic, all active applications running `Chat` receive that message as subscribers to that topic. This is the most basic form of publish and subscribe activity.

◆ To start Chat sessions:

1. Open a console window to the `TopicPubSub\Chat` folder, then enter:

   `Chat -u Chatter1`

   This command starts a `Chat` session for the user, **`Chatter1`**.

2. Open another console window to the `TopicPubSub\Chat` folder, then enter: **`Chat -u Chatter2`**

   This command starts a `Chat` session for the user, **`Chatter2`**.

◆ To Chat:

1. In one of the **`Chat`** windows, type any text and then press **Enter**. The text is displayed in both **`Chat`** windows, preceded by the name of the user that initiated that text.

2. In the other **`Chat`** window, type text and then press **Enter**. The text is displayed in both **`Chat`** windows preceded by that username.

The `Chat` sample shows inter-application asynchronous communications. If subscribers miss some of the messages, they just pick up the latest messages whenever they reconnect to the broker. Nothing is retained and nothing is guaranteed to be delivered, so throughput is fast.

Talk Application (PTP)

In the `Talk` application, whenever a text message is sent to a given queue, all active `Talk` applications are waiting to receive messages on that queue, taking turns as the sole receiver of the message at the front of the queue.

◆ To start Talk sessions:

The first `Talk` session receives on the first queue and sends to the second queue, while the other `Talk` session does the opposite.

1. Open a console window to the `QueuePTP\Talk` folder, then enter:

   `Talk -u Talker1 -qr SampleQ1 -qs SampleQ2`

   This command starts a `Chat` session for the user, **`Talker1`**.

2. Open another console window to the `QueuePTP\Talk` folder, then enter:

   `Talk -u Talker2 -qr SampleQ2 -qs SampleQ1`

   This command starts a `Chat` session for the user, **`Talker2`**.

◆ To Talk:

1. In the **`Talker2`** window, type any text and then press **Enter**.

   The text is displayed in only the **`Talker1`** window, preceded by the name of the user who sent the message.

2. In the **`Talker1`** window, type text and then press **Enter**.

   The text is displayed in only the **`Talker2`** window, preceded by the username of the sender.

Reviewing the Chat and Talk Samples

You can continue exploring these samples by opening several windows:

- **Chat** — If you run several **`Chat`** windows, every window displays the message, including the publisher. You can modify the source code to suppress delivery of a `Chat` message to its publisher. That Pub/Sub broadcast characteristic can be stopped with a `noLocal` parameter on the **`createSubscriber`** method. In this case, each subscriber receives everyone else's messages, but not its own.
- **Talk** — If you run several **`Talk`** windows, you will still see only one receiver for any message. Under **`Talk`** (PTP), there is only one receiver. Start two more **`Talker`** windows (**`Talker3`** and **`Talker4`**), then use the **`Talker1`** window to send **`1`** through **`9`**, each as a message. For example, enter the following:

  `1` **Enter**, **`2`** **Enter**, ..., **`9`** **Enter**

  Notice how two receivers take turns receiving the messages. Because queue receivers have a prefetch mechanism, the talkers might show that **`Talker1`** receives messages **`1`**, **`2`**, and **`3`**, **`Talker2`** receives messags **`4`**, **`5`**, and **`6`**, and then **`Talker1`** receives **`7`**, **`8`**, and **`9`**.

Samples of Additional Message Types

Most of the SonicMQ samples use the `TextMessage` type because they accept user input in the console windows. Additional message type samples demonstrate how `Map` messages and `XML` messages are handled.

Map Messages (PTP)

The Map message type transfers a collection of assigned names and their respective values. The names and values are assigned by `set( )` methods for the C# primitive data type of the value. The `MapMessage` name-value pairs are sent in the message body. For example:

```
mapMessage.setInt("FiscalYearEnd", 10)
mapMessage.setString("Distribution", "global")
mapMessage.setBoolean("LineOfCredit", true)
```

You can extract the data from the received message in any order. Use a `get( )` method to cast a data value into an acceptable data type. For example:

```
mapMessage.getShort("FiscalYearEnd")
mapMessage.getString("Distribution")
mapMessage.getString("LineOfCredit")
```

◆ To start MapTalk sessions:

This example starts two `MapTalk` sessions, one for an accounting group and one for an auditing group. The first `MapTalk` session receives on the first queue and sends to the second queue, while the other session does the opposite.

1. Open a console window to the `QueuePTP\MapTalk` folder, then enter:
   `MapTalk -u QAccounting -qr SampleQ1 -qs SampleQ2`
   This command starts a `MapTalk` session for the user **`QAccounting`**.
2. Open another console window to the `QueuePTP\MapTalk` folder then enter:
   `MapTalk -u QAuditing -qr SampleQ2 -qs SampleQ1`
   This command starts a `MapTalk` session for the user **`QAuditing`**.

◆ To send and receive MapMessages:

❖ In the **`QAccounting`** window, type text then press **Enter**.

The message sender packages two items: the username as the String `sender` and the text input as the String `content`, as shown in the source code of the sample `MapTalk.cs`:

```
Sonic.Jms.MapMessage msg = sendSession.createMapMessage();
msg.setString("sender", username);
msg.setString("content", s);
```

The message receiver casts the message as a `MapMessage`. If that casting is unsuccessful, `MapTalk` reports that an invalid message arrived. The `MapMessage` is decomposed and displayed as shown in the source code of the sample `MapTalk.cs`:

```
String sender  = mapMessage.getString("sender");
String content = mapMessage.getString("content");
System.Console.WriteLine(sender + ": " + content);
```

Decomposing Multipart Messages

Multipart messages are familiar files in mail applications—pictures, documents, text, and executable files all packaged as attachments to a mail message. Multipart messages are also used in Business-to-business applications that use of the Simplified Object Access Protocol (SOAP) 1.1 with Attachments.

The sample assigns each component to a message part, then sends the message with its list of parts. The receiver reverses the process to isolate each message part.

- **To run the Multipart message sample:**
  - Open a console window to the `QueuePTP\MultipartMessage`, then enter:
    `Multipart -u aUser`

    This command starts a **`Multipart`** session for the user **`aUser`**.

    The sample demonstrates creating and assembling the parts of a message into a single multipart message, as shown in the following output to the console window:

```
sending part1..a TextMessage
```

The multipart message is sent as an instance of `MultipartMessage`. The receiver of the message discovers that the message is multipart, how many parts it contains (in this example, only one), and goes through a process of disassembling the parts, as shown in the following output to the console window:

```
    received MutipartMessage....
    ******* Beginning of MultipartMessage ******
    Extend_type property = x-sonicmq-multipart
    partCount of this MultipartMessage = 1
    --------Beginning of part 1

Part.contentType = application/x-sonicmq-textmessage
Part.contentId = CONTENTID1
content in TextMessage...  this is a JMS TextMessage
    --------end of part 1
```

When a part is complete, the receiving application can act on that part. The message parts should be handled in a transactional way so that the messages parts can be rolled back if the process fails before it completes all its parts.

Reviewing the Additional Message Type Samples

The samples demonstrated in this section show:

- The message type characteristics are identical in PTP and Pub/Sub.
- These messages are limited to capturing a single chunk of text in the console window.
- These messages use the `is` operator to identify and cast the message into a `MapMessage`.

You can modify the source code of these samples to set some map values to C# primitives in the `MapMessage` and then get the map values, coercing them into acceptable data types.

See the exercises in “Enhancing the Basic Samples” on page 70 that describe these changes. See also “Message Type” on page 141.

Transaction Samples

Transacted messages are a group of messages that form a single unit of work. Much like an accounting transaction made up of a set of balancing entries, a messaging example might be a set of financial statistics where each entry is a completely formed message and the full set of data comprises the update.

A session is declared **transacted** when the session is created. While producers—PTP senders and Pub/Sub publishers—produce messages as usual, the messages are stored at the broker until the broker is notified to act on the transaction by delivering or deleting the messages. The programmer must determine when the transaction is complete:

- Call the method to **commit** the set of messages. The session **`commit( )`** method tells the broker to sequentially release each of the messages that have been cached since the last transaction. In this sample, the commit case is set for the string **`OVER`**.
- Call the method to **roll back** the set of messages. The session **`rollback( )`** method tells the broker to flush all the messages that have been cached since the last transaction ended. In this sample, the rollback case is set for the string **`OOPS!`**.

TransactedTalk Application (PTP)

The following procedures explain how to run the `TransactedTalk` sample application.

◆ To start TransactedTalk sessions:

The first `TransactedTalk` session receives on the first queue and sends to the second queue, while the other session does the opposite.

1. Open a console window to the `QueuePTP\TransactedTalk` folder, then enter:
   `TransactedTalk -u Accounting -qr SampleQ1 -qs SampleQ2`
   This command starts a **`TransactedTalk`** session for the user **`Accounting`**.
2. Open another console window to the `QueuePTP\TransactedTalk` folder, then enter:
   `TransactedTalk -u Operations -qr SampleQ2 -qs SampleQ1`
   This command starts a **`TransactedTalk`** session for the user **`Operations`**.

◆ To build a PTP transaction and commit it:

1. In a **`TransactedTalk`** window, type any text and then press **Enter**.
   Notice that the text is ***not*** displayed in the other **`TransactedTalk`** window.
2. Type more text in that window and then press **Enter**.
   The text is still not displayed in the other **`TransactedTalk`** window.
3. Type **`OVER`** and then press **Enter**.
   The **`TransactedTalk`** window in which you are working displays the message:
   ```
   Committing messages...Done
   ```
   All the messages you sent to a queue are delivered to the receiver. Subsequent entries will form a new transaction.

◆ To build a PTP transaction and roll it back:

1. In one of the **`TransactedTalk`** windows, type text and then press **Enter**.
2. Type more text in that window and then press **Enter**.
3. Type **`OOPS!`** and then press **Enter**. Nothing is published.
   The **`TransactedTalk`** window in which you are working displays the message:
   ```
   Cancelling messages...Done!
   ```
   All messages are removed from the broker. Subsequent messages form a new transaction. Any messages you resend are redelivered.

TransactedChat Application (Pub/Sub)

The following procedures explain how to run the `TransactedChat` sample application.

◆ To start Pub/Sub TransactedChat sessions:

1. Open a console window to the `TopicPubSub\TransactedChat` folder, then enter:
   `TransactedChat -u Sales`

   This command starts a **`TransactedChat`** session for the user **`Sales`**.

2. Open another console window to the `TopicPubSub\TransactedChat` folder, then enter:
   `TransactedChat -u Audit`

   This command starts a **`TransactedChat`** session for the user **`Audit`**.

◆ To build a Pub/Sub transaction and commit it:

1. In the **`Sales`** window, type any text and then press **Enter**.

   Notice that the text is ***not*** displayed in the **`Audit`** window.

2. Type more text in the **`Sales`** window and then press **Enter**.

   The text is still not displayed in the **`Audit`** window.

3. Type **`OVER`** and then press **Enter**.

   The **`TransactedChat`** window in which you are working displays the message:

   `Committing messages...Sales:`*`Message_Text`*

   All of the messages now display in sequence in the **`Audit`** window. All of the lines you published to a topic are delivered to subscribers. Subsequent entries form a new transaction.

◆ To build a Pub/Sub transaction and roll it back:

1. In the **`Sales`** window, type text and then press **Enter**.

2. Type more text in that window and then press **Enter**.

3. Type **`OOPS!`** and then press **Enter**.

   The **`TransactedTalk`** window in which you are working displays the message:

   `Cancelling messages...Done!`

   No messages are published. All messages are removed from the broker. Subsequent entries form a new transaction. Any messages you resend are redelivered.

Reviewing the Transaction Samples

The transaction samples show:

- The transaction scope is between the client in the session and the broker. When the broker receives commitment, the messages are placed onto queues or topics in the order in which they were buffered but with no transaction controls. Message delivery is normal:
  - **PTP Messages** — The order of messages in the queue is maintained with adjustments for priority differences but there is no guarantee that—when multiple consumers are active on the queue—a `MessageConsumer` will receive one or more of the `MessageProducer`'s transacted messages.
  - **Pub/Sub Messages** — Messages are delivered in the order entered in the transaction, yet are influenced by the priority setting of these and other messages, the use of additional receiving sessions, and the use of additional or alternate topics. The messages are not delivered as a group.
- A transaction is a set of messages that is complete only when a command is given. As an alternative, message volume could be reduced by packaging sets of messages into a single multipart message.
- While most of the samples use two sessions—a producer session to listen for keyboard input and send messages, and a consumer session to listen for messages and receive them—the transacted samples set only the producer session as transacted so that committing or rolling back impacts only the sent messages.

  Changing the consumer behavior has no real effect on nondurable Pub/Sub messages but causes an interesting behavior in PTP: when you roll back receipt of messages, the message listener sees the messages again and then simply receives them again. Rolling back a transacted consumer session causes the messages to be redelivered.

  For more information, see "Transacted Sessions" on page 123.

Reliable, Persistent, and Durable Messaging Samples

The preceding applications make the same delivery promise: if you are connected and receiving during the message's lifespan, you can be a consumer of this message.

One of the features of SonicMQ is the breadth of services that can be applied to messaging to give just the right **quality of service (QoS)** for each type and category of message.

There are programmatic mechanisms for:

- Increasing the chances that the client and broker are actively connected
- Registering a PTP sender's interest in routing messages that are undeliverable to a dead message queue (DMQ) and sending notification events to the administrator
- Registering a Pub/Sub subscriber's interest in messages published to a topic even when the subscriber is disconnected

The reliable, persistent, and durable messaging samples demonstrate these features of SonicMQ.

Reliable Connections

The `ReliableTalk` and `ReliableChat` samples demonstrate techniques for monitoring a connection for exceptions and re-establishing the connection if it is dropped.

Note The Reliable samples use an aggressive technique (**CTRL+C**) that emulates an unexpected broker interruption.
An intentional shutdown invokes an administrative **Shutdown** function on the broker. This function is a command in the Sonic Management Console runtime.

In a `Talk` session, if the broker stopped and you sent a message, you see:

```
Sonic.Jms.IllegalStateException: The session is closed
```

This error occurs because the `Talk` sample assumes that the connection is established and available. The `Talk` sample does not consider the possibility that a problem occurred with the connection (such as the network or the broker failing).

The `ReliableTalk` and `ReliableChat` samples, in contrast, are written to handle exceptions. Both samples use a connection setup routine for retrying connections that fail for some reason.

The `ReliableTalk` and `ReliableChat` samples also use the **`PERSISTENT`** delivery mode option to ensure that messages are logged before they are acknowledged and are nonvolatile in the event of a broker failure. Consequently, as shown in the `ReliableTalk` example, the application tries repeatedly to reconnect.

A unique SonicMQ feature monitors the heartbeat of the broker by pinging the broker at a preset interval, letting the thread sleep for a while but initiating reconnection if the broker does not respond. For more information, see "Creating and Monitoring a Connection" on page 90.

These examples demonstrate techniques an application programmer can use to explicitly handle connection exceptions. These samples do not, however, take advantage of an important SonicMQ feature—**fault-tolerant connections**.

Fault-tolerant connections automatically detect problems with a connection and seamlessly reconnect, if possible, either to the same broker or possibly to a backup broker (if your deployment is set up to perform broker replication). This feature significantly enhances the reliability of a connection.

The exception handling logic in the `ReliableTalk` and `ReliableChat` programs is devoted to retrying a connection after the connection fails for some reason. This logic, as written, is not necessary with a fault-tolerant connection, because the fault-tolerant connection automatically retries the connection on your behalf. A fault-tolerant connection can attempt to reconnect indefinitely or for a fixed period of time, depending on how it is configured.

When a fault-tolerant connection encounters a problem and is able to reconnect, your application does not receive an exception and continues processing after the connecting is reestablished.

When a fault-tolerant connection times out without successfully reconnecting, the connection is dropped and an exception is generated. Your exception handling logic can decide what to do with the exception. Retrying the connection might not make sense if the automatic retry was unsuccessful.

See "Fault-Tolerant Connections" on page 93.

ReliableTalk Application (PTP)

The following procedure explains how to run the `ReliableTalk` sample application.

◆ To run the ReliableTalk sample:

1. Open a console window to the `QueuePTP\ReliableTalk` folder, then enter:
   `ReliableTalk -u AlwaysUp -qr SampleQ1 -qs SampleQ1`
   This command starts a `ReliableTalk` session for the user **`AlwaysUp`**.

2. Type some text, then press **Enter**.
   The text is displayed, preceded by the user name that initiated that text. The message was sent from the client application to the `SampleQ1` queue on the broker and then returned to the client as a receiver on that queue. The connection is active.

3. Stop the broker by pressing **Ctrl+C** in the broker window.
   The connection is broken. The `ReliableTalk` application tries repeatedly to reconnect:

```
[ MESSAGE RECEIVED ] AlwaysUp: Hello


There is a problem with the connection.
   JMSException: end of stream
Please wait while the application tries to re-establish the connection...
Attempting to create connection...
Failed to connect to broker: localhost:2506 : I/O Error [connect: No
connection could be made because the target machine actively refused it]
Pausing 10 seconds before retry.
Attempting to create connection...
Failed to connect to broker: localhost:2506 : I/O Error [connect: No
connection could be made because the target machine actively refused it]
Pausing 10 seconds before retry.
Attempting to create connection...
```

4. Restart the container and broker by using the Windows **Start** menu command. The **`ReliableTalk`** application reconnects:

```
Attempting to create connection...
...Connection created.
...Setup complete.
...Connection started.

Receiving messages on queue "SampleQ1".


Enter text to send to queue "SampleQ1".
Press Enter to send each message.
```

ReliableChat Application (Pub/Sub)

The following procedure explains how to run the `ReliableChat` sample application.

◆ To run the ReliableChat sample:

1. Open a console window to the `TopicPubSub\ReliableChat` folder, then type:

   `ReliableChat -u AlwaysUp`

   This command starts a `ReliableChat` session for the user **`AlwaysUp`**.

2. Type text and then press **Enter**.

   The text is displayed, preceded by the user name that initiated that text. The message was sent from the client application to the broker and then returned to the client as a subscriber to that topic. The connection is active.

3. Stop the broker by pressing **Ctrl+C** in the broker window.

   The connection is broken. The `ReliableChat` application tries repeatedly to reconnect. The console window shows message similar to those in the `ReliableTalk` example.

   Restart the container and broker by using its Windows **Start** menu command or the `startmf` script. The `ReliableChat` application reconnects. The console window shows message similar to those in the `ReliableTalk` example.

DurableChat Application (Pub/Sub)

In Pub/Sub messaging, when messages are produced, they are sent to all active consumers who subscribe to a topic. Some subscribers register an enduring interest in receiving messages sent while they were inactive. These **durable subscriptions** are permanent records in the broker's persistent storage mechanism.

Whenever a subscriber reconnects to the topic (under the registered username, subscriber name, and client identifier), all undelivered messages to that topic that have not expired are delivered immediately. The administrator can terminate durable subscriptions or a client can use the `unsubscribe( )` method to close the durable subscription.

In an application, there are only a few changes to set up a subscriber as a durable subscriber. Where `Chat` is coded as:

```
subscriber = subSession.createConsumer(topic);
```

`DurableChat` is coded as follows:

```
//Durable Subscriptions index on username, clientID, subscription name
//It is a good practice to set the clientID:
connection.setClientID(CLIENT_ID);
...
subscriber = subSession.createDurableSubscriber(topic, "SampleSubscription");
```

As with `ReliableChat`, using the **`PERSISTENT`** delivery mode ensures that messages are logged before they are acknowledged and are nonvolatile in the event of a broker failure.

Figure 4 shows what occurs when the subscriber requests an extra effort to ensure delivery.

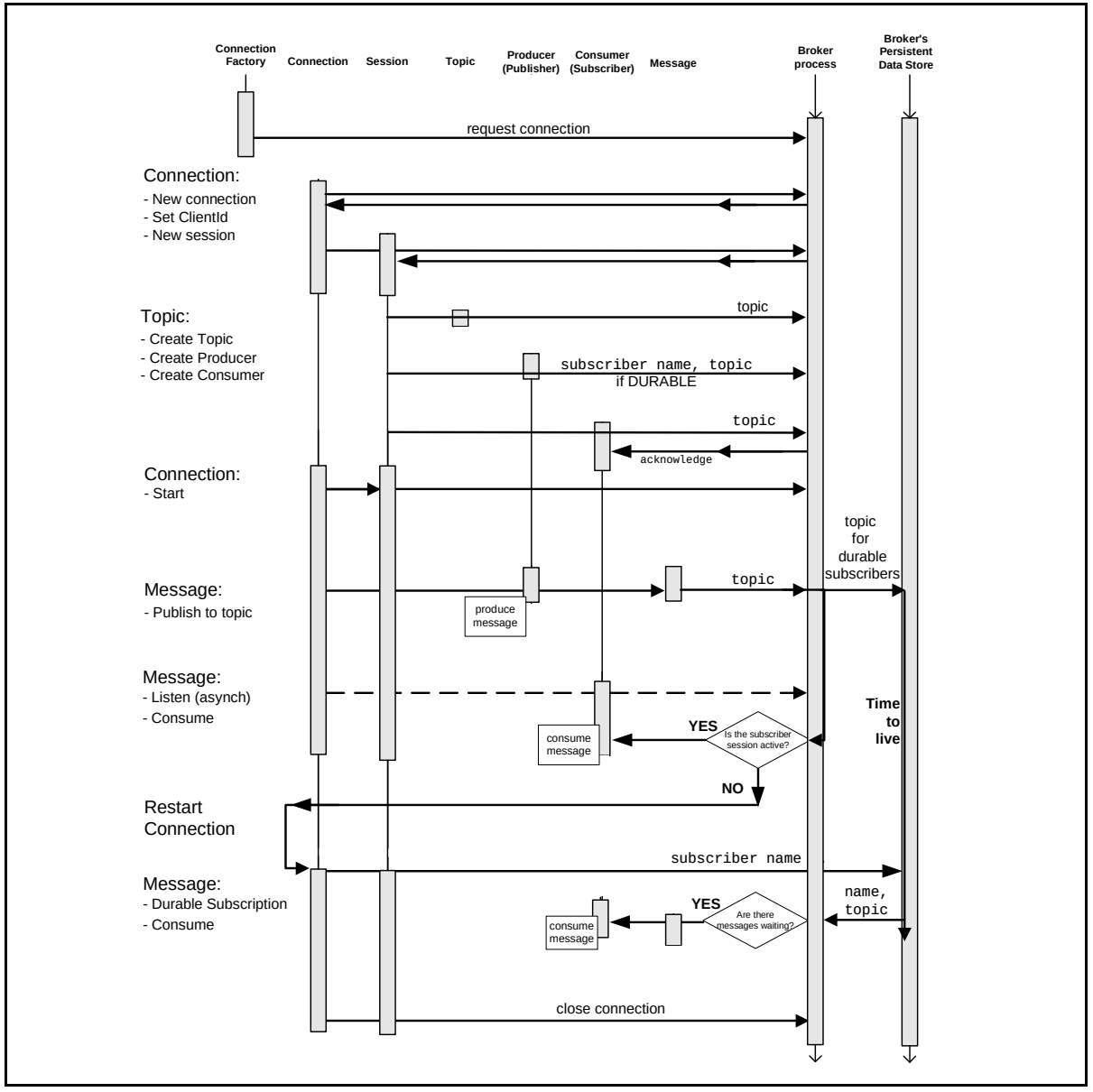


Figure 4. Sequence Diagram for the DurableChat Application

◆ To run the DurableChat sample:

1. Open a console window to the `TopicPubSub\DurableChat` folder, then enter:
   `DurableChat -u AlwaysUp`
   This command starts a **`DurableChat`** session for the user **AlwaysUp**.
2. Open another console window to the `TopicPubSub\DurableChat` folder, then enter:
   `DurableChat -u SometimesDown`
   This command starts a **`DurableChat`** session for the user **`SometimesDown`**.
3. In the **`AlwaysUp`** window, type text and then press **Enter**.
   The text is displayed on both subscriber consoles.
4. In the **`SometimesDown`** window, type text and then press **Enter**.
   The text is displayed on both subscriber consoles.
5. Stop the **`SometimesDown`** session by pressing **Ctrl+C**.
6. In the **`AlwaysUp`** window, send one or more messages.
   The text is displayed on that subscriber's console.
7. In the window where you stopped the `DurableChat` session, restart the session under the same name.
   When the `DurableChat` session reconnects, the retained messages are delivered and then displayed in the **`SometimesDown`** console window.

Although the messages are durable, they do not last indefinitely. The publisher of the message sets a **time-to-live** parameter—a value that, when added to the publication timestamp, determines the expiration time of the message. The time-to-live value in milliseconds can be any positive integer. In this sample, the time-to-live is 1,800,000 milliseconds (thirty minutes). Setting the value to zero retains the message indefinitely.

Reviewing Reliable, Persistent, and Durable Messaging

The characteristics demonstrated in this section improve Quality of Service (QoS) while requiring modest overhead. The examples in this section can be combined to create a reliable, persistent talk and a reliable, durable chat. The source code of these samples is readily transferable into your applications.

There are also other facets to consider for optimal QoS, including the various security, encryption, access control, and transport protocols. See the *Progress SonicMQ Deployment Guide* for information about security and protocols.

Request and Reply Samples

Loosely coupled applications require special techniques when it is important for the publisher to certify that a message was delivered in either messaging domain:

- **Point-to-point** — While a sender can see if a message was removed from a queue, implying that it was delivered, there is no indication where it went.
- **Publish and Subscribe** — While the publisher can send long-lived messages to durable receivers and receive acknowledgement from the broker, neither of these techniques confirms that a message was actually delivered or how many, if any, subscribers received the message.

A message producer can request a reply when a message is sent. A common way to do this is to set up a **temporary destination** and header information that the consumer can use to create a reply to the sender of the original message.

In both Request and Reply samples, the replier's task is a simple data processing exercise. The task is to standardize the case of the text sent—receive text and send back the same text as either all uppercase characters or all lowercase characters—and then publish the modified message to the temporary destination that was set up for the reply.

While request-and-reply provides proof of delivery, it is a blocking transaction—the requestor waits until the reply arrives. While this situation might be appropriate for a system that, for example, issues lottery tickets, it might be preferable in other situations to have a formally established return destination that echoes the original message and a **correlation identifier**—a designated identifier that certifies that each reply is referred to its original requestor.

Note You can use the **`JMSReplyTo`** and **`JMSCorrelationID`** message header fields as a suggested design pattern for correlation identification. The application programmer ultimately decides how these fields are used, if at all. See "Message Header Fields" on page 152.

The sample applications use the classes **`TopicRequestor`** and **`QueueRequestor`**. You should create the Request/Reply helper classes that are appropriate for your application.

Request and Reply (PTP)

In the PTP domain, the requestor application can be started and even send a message before the replier application is started. The queue holds the message until the replier is available. The requestor is still blocked, but when the replier's message listener receives the message, it releases the blocked requestor. The sample code includes an option (**`-m`**) to switch the mode between uppercase and lowercase.

◆ To start the PTP Request and Reply sessions:

1. Open two console windows to the `QueuePTP\RequestReply` folder.
2. In one console window enter:

   `Requestor -u QRequestor`

   This command starts a PTP `Requestor` session for the user **QRequestor**.
3. In the other console window enter:

   `Replier -u QReplier`

   This command starts a PTP `Replier` session for the user **`QReplier`**.

   The default value of the mode in this sample is `uppercase`.

◆ To test a PTP request and reply:

❖ In the **`Requestor`** window, type **`AaBbCc`**, then press **Enter**.

The **`Replier`** window reflects the activity, displaying:

```
[Request] QRequestor: AaBbCc
```

The `Replier` does its operation (converts text to uppercase) and sends the result in a message to the `Requestor`. The **`Requestor`** window receives the reply from the `Replier`:

```
[Reply] Uppercasing-QRequestor: AABBCC
```

Request and Reply (Pub/Sub)

In this example in the Pub/Sub domain, the replier application must be started before the requestor so that the Pub/Sub replier's message listener can receive the message and release the blocked requestor.

◆ To start the Pub/Sub Request and Reply sessions:

1. Open two console windows to the `TopicPubSub\RequestReply` folder.
2. In one of the windows enter:

   `Replier`

   This command starts a Pub/Sub `Replier` session.

   The default value of the mode in this sample is `uppercase`.
3. In the other window enter:

   `Requestor`

   This command starts a Pub/Sub `Requestor` session.

◆ To test a Pub/Sub request and reply:

❖ In the **`Requestor`** window, type **`AaBbCc`**, then press **Enter**.

The **`Replier`** window reflects the activity, displaying:

```
[Request] SampleReplier: AaBbCc
```

The replier completes its operation (converts text to uppercase) and sends the result in a message to the requestor. The requestor receives the reply from the replier:

```
[Reply] Uppercasing-SAMPLEREQUESTOR: AABBCC
```

Reviewing the Request and Reply Samples

These request and reply samples show:

- Request and reply mechanisms are very similar across domains.
- While there might be zero or many subscriber replies, there is, at most, one PTP reply.
- Using message header fields (`JMSReplyTo` and `JMSCorrelationID`) and the requestor sample classes (`Sonic.Jms.TopicRequestor` and `Sonic.Jms.QueueRequestor`) are suggested implementations. However, you can accomplish request-reply functionality using other means (see "Reply-to Mechanisms" on page 188).

Selection and Wild Card Samples

While specific queues and topics provide focused content nodes for messages that are of interest to an application, there are circumstances where the programmer might want to qualify the scope of interest a consumer has in messages, as with an SQL `WHERE` clause.

Conversely there are circumstances where the specificity of having to declare each topic of interest becomes slow and unwieldy. Because topic names can be created as needed (assuming there are no security constraints), a subscriber application might need to scan many topics.

These situations are contrasted in these samples:

- If you force too much traffic into a small number of destinations and then use selector strings, performance is impacted.
- If you use many topic names, SonicMQ's hierarchical topic structure bypasses much message selector overhead. You can apply wild cards to subscriptions by just subscribing to parent topic nodes.

In PTP domains, all message selection takes place on the server. However, in Pub/Sub domains, all messages for a subscribed topic are, by default, delivered to the subscriber and then the filter is applied to decide what is consumed. When the subscriber message traffic is a burden and server resources can handle it, you can cause a Pub/Sub message selector to filter the messages on the broker by calling `factory.`**`setSelectorAtBroker`**`(true)` on the `ConnectionFactory`.

SelectorTalk Application (PTP)

The `SelectorTalk sample` application starts by declaring a selector *`String-value`* to be attached to the message as `PROPERTY_NAME='`*`String_value`*`'`. The sessions send and receive to alternate queues so that they pass each other messages. The receive method has a selector string parameter (**`-s`**). In PTP domains, all messages for a queue topic are filtered on the broker and then the qualified messages are delivered to the consumer.

◆ To run SelectorTalk sessions:

1. Open a console window to the `QueuePTP\SelectorTalk` folder, then enter:
   `SelectorTalk -u AAA -s North -qr SampleQ1 -qs SampleQ2`

   This command starts a `SelectorTalk` session for the user **`AAA`** with the selector string **`North`**.

2. Open another console window to the `QueuePTP\SelectorTalk` folder, then enter:
   `SelectorTalk -u BBB -s South -qr SampleQ2 -qs SampleQ1`
   This command starts a `SelectorTalk` session for the user **`BBB`** with the selector string **`South`**.
3. In the **`AAA`** window, type any text and then press **Enter**.
   The message is enqueued but there is no receiver. The **`BBB`** selector string does not see any enqueued messages except those that evaluate to **`South`**.
4. Stop the **`BBB`** session by pressing **Ctrl+C**.
5. In the **`BBB`** window start a new session, changing the selector string:
   `SelectorTalk -u BBB -s North -qr SampleQ2 -qs Sample q1`
   The session starts and the message that was enqueued is immediately received.
6. In the **`AAA`** window, again type any text and then press **Enter**.
   The message is enqueued and the **`BBB`** selector string qualifies the message for immediate delivery.

SelectorChat Application (Pub/Sub)

In the `SelectorChat` application, the application starts by declaring the *`String-value`* that is attached to the message as `PROPERTY_NAME='String_value'`. The method for the subscription declares the sample's topic, `jms.samples.chat`, and the selector string (**`-s`**).

◆ To run SelectorChat sessions:

1. Open a console window to the `TopicPubSub\SelectorChat` folder, then enter:
   `SelectorChat -u Closer -s Sales`
   This command starts a `SelectorChat` session for the user **`Closer`** with the selector string **`Sales`**.
2. Open another console window to the `TopicPubSub\SelectorChat` folder, then enter:
   `SelectorChat -u Presenter -s Marketing`
   This command starts a `SelectorChat` session for the user **`Presenter`** with the selector string **`Marketing`**.
3. In the **`Closer`** window, type any text and then press **Enter**.
   The text is only displayed in the **`Closer`** window. The **`Presenter`** selector string excludes the **`Sales`** message.

4. In the **Presenter** window, type any text and then press **Enter**.

   The text is only displayed in the **Presenter** window. The **Closer** selector string excludes the **Marketing** message.

5. Stop the **Closer** session by pressing **Ctrl+C**.

6. In the **Closer** window start a new session, changing the selector string:

   ```
   SelectorChat -u Closer -s Marketing
   ```

7. Type text in either window and then press **Enter**.

   Because the selector string matches for the sessions, the text is displayed in both windows.

HierarchicalChat Application (Pub/Sub)

SonicMQ provides a hierarchical topic structure that allows wild card subscriptions. This feature enables an application to have the power of a message selector plus a more streamlined way to often get the same result. In this sample, each application instance creates two sessions, one for the publish topic (**-t**) and one for the subscribe topic (**-s**).

◆ To start HierarchicalChat sessions:

1. Open a console window to the `TopicPubSub\HierarchicalChat` folder, then enter:

   ```
   HierarchicalChat -u HQ -t sales.corp -s sales.*
   ```

   This command starts two `HierarchicalChat` sessions for the user **HQ**:

   - One session that publishes messages to the topic **sales.corp**
   - One session that listens for messages from the subscribe topic **sales.***

2. Open another console window to the `TopicPubSub\HierarchicalChat` folder then enter:

   ```
   HierarchicalChat -u America -t sales.usa -s sales.usa
   ```

   This command starts two `HierarchicalChat` sessions for the user **America**:

   - One session that publishes messages to the topic **sales.usa**
   - One session that listens for messages from the subscribe topic **sales.usa**

◆ **To run HierarchicalChat:**

1. In the **HQ** window, type text and then press **Enter**.

   The text is displayed in only the **HQ** window because **HQ** subscribes to all topics in the sales hierarchy while **America** is subscribing to only the **sales.usa** topic.

2. In the **America** window, type text and then press **Enter**.

   The text is displayed in both windows because:

   - **America** subscribes to the **sales.usa** topic.
   - **HQ** subscribes to all topics that start with **sales.**

Reviewing the Selection and Wild Card Samples

While selector strings can provide a variety of ways to qualify what messages are sent to a consumer, the overhead in the evaluation of the selectors can impact overall system performance. `HierarchicalChat` illustrates a feature of SonicMQ that provides the advantages of selectors with minimal overhead. Note also that security access control uses similar wild card techniques to enable read/write security for all subtopics within a topic node.

Another way to increase specificity is to use complex SQL statements. For information on hierarchical security, including hierarchical name spaces and security, see the *Progress SonicMQ Deployment Guide*.

Test Loop Sample: QueueRoundTrip Application (PTP)

A simple loop test lets you experiment with messaging performance. The `RoundTrip` sample application sends a brief message to a sample queue and then uses a temporary queue to receive the message back. A counter is incremented and the message is sent for another trip. After completing the number of cycles you entered when you started the test, the run completes by displaying summary and average statistics.

Note Use this sample to evaluate features only; it is not intended as a performance tool. For information on performance, see the *Progress SonicMQ Performance Tuning Guide*.

◆ To run QueueRoundTrip:

1. Open a console window to the `QueuePTP\QueueRoundTrip` folder, then enter:

   `QueueRoundTrip -n 100`

   This command starts a `QueueRoundTrip` session that sends a message on `100` round trips to a temporary queue.

   The `QueueRoundTrip` window displays information about the cycles:

   ```
   Sending Messages to Temporary Queue...

   Time for 100 sends and receives:                631ms
   Average Time per message:                       6.31ms

   Press enter to continue...
   ```

2. In the `QueueRoundTrip` window, enter:

   `QueueRoundTrip -n 1000`

   This command starts a `QueueRoundTrip` session that sends a message on `1000` round trips to a temporary queue.

   The `QueueRoundTrip` window displays information for the 1000 cycles:

   ```
   Sending Messages to Temporary Queue...

   Time for 1000 sends and receives:               5538ms
   Average Time per message:                       5.538ms

   Press enter to continue...
   ```

Enhancing the Basic Samples

After exploring the basic samples you can modify the sample source files to learn more about SonicMQ.

Building the Progress SonicMQ Sample .NET Programs

In order to enhance the samples, you need to edit the sample source files and then compile the saved results.

The following steps build the sample programs.

◆ To build Progress SonicMQ .NET Client samples:

1. Download and install the appropriate .NET Framework SDK and Visual Studio software from Microsoft.
2. Down load and install the appropriate NAnt build software and related utilities from `nant.sourceforge.net`.
3. Open a console window to *`net_client_install_dir`*`/samples`
4. Set `NAnt.exe`, `cl.exe`, `link.exe` and `csc.exe` on the path.
5. Run:

   ```
   nant -buildfile:samples.build
   ```

Use Common Topics Across Clients

When you run the Pub/Sub samples you might notice that while all the `Chat` applications receive `Chat` messages and all the `DurableChat` applications receive `DurableChat` messages, they do not receive each other's messages. This is because the applications are publishing to different topics. You can set the two applications to monitor messages on the same topic.

◆ To put Chat and DurableChat on the same topic:

1. Open the SonicMQ sample file `DurableChat.cs` for editing.
2. Change the value of the variable `APP_TOPIC` from `jms.samples.durablechat` to `jms.samples.chat.`
3. Save and compile the edited `DurableChat.cs` file.
4. Run the new `DurableChat.exe` file.

   Messages sent from `DurableChat` and `Chat` are received by both regular and durable subscribers. The durable subscribers will receive messages when they recover from offline situations, but the regular subscribers will not recover missed messages.

Important If you make this change, the broker will maintain the durable subscriptions for all the `Chat` messages. While `DurableChat` messages expire after 30 minutes, `Chat` messages are published with the default time-to-live (never expire). The `Chat` messages endure for durable subscribers until one of the following occurs:

- The durable subscriber connects to receive the messages
- The durable subscriber explicitly unsubscribes
- The persistent storage mechanism is initialized

Trying Different RoundTrip Settings

With the `RoundTrip` sample application, you choose a number of produce-then-consume iterations to perform when the application runs. You can enhance the application to explore the time impact of other settings and parameters as well.

Note Use this sample to evaluate features only. It is not intended as a performance tool. For information on performance, see the *Progress SonicMQ Performance Tuning Guide*.

A counter is incremented and the message is sent for another trip. After completing the number of cycles you entered when you started the test, the run completes by displaying summary and average statistics.

◆ To extend the QueueRoundTrip sample:

1. Edit the sample file `QueuePTP\QueueRoundTrip.cs` to establish any of the following behavior changes:
   - Change the `Sonic.Jms.Ext.DeliveryMode` from **`NON_PERSISTENT`** to **`PERSISTENT`**. Run it, then change it to **`NON_PERSISTENT_ASYNC`**.
   - You can change the `priority` or `timeToLive` values, but in this sample the effect would be negligible.
   - Change the message type from the bodyless `createMessage( )` to a bodied message type, such as `createTextMessage( )`.
   - Create a set of sample strings (or other appropriate data type) to populate a bodied-message payload with different size payloads.
   - Use `createXMLMessage( )` and load the message payload with well-formed XML data. Then try the same payload as a `TextMessage`.
   - Change the receiver session acknowledgement mode from **`AUTO_ACKNOWLEDGE`** to **`DUPS_OK_ACKNOWLEDGEMENT`**. Change it again to **`CLIENT_ACKNOWLEDGE`** or **`SINGLE_MESSAGE_ACKNOWLEDGE`**, then add an explicit `acknowledge( )` after the receive is completed.
2. Save and compile the edited `.cs` file.
3. Open a console window to the `QueuePTP\QueueRoundTrip` folder, then enter

   `QueueRoundTrip -n 100`
4. Observe the results and compare them to other round trips (see “Test Loop Sample: QueueRoundTrip Application (PTP)” on page 69).

Modifying the MapMessage to Use Other Data Types

The `MapMessage` sample application is limited as its content is just a snippet of text. The key concepts of the `MapMessage` sample are that:

- The body is a collection of name-value pairs.
- The values can be C# primitives.
- The receiver can access the names in any sequence.
- The receiver can attempt to coerce a value to another data type.

The following exercise adds some mixed data types to the `MapTalk` source file before the message is sent. Then the receiver takes the data in a different sequence and formats it for display.

The example uses typed **`set( )`** methods to populate the message with *`name-typedValue`* pairs. The **`get( )`** methods retrieve the named properties and attempt coercion if the data type is dissimilar.

◆ To extend the MapTalk sample to use and display other data types:

1. Edit the sample file `MapTalk.cs` at these lines:

```
Sonic.Jms.MapMessage msg = sendSession.createMapMessage();
msg.setString("sender", username);
msg.setString("content", s);
```

2. Add the lines for the **`set( )`** methods (or similar lines):

```
msg.setInt("FiscalYearEnd", 10);
msg.setString("Distribution", "global");
msg.setBoolean("LineOfCredit", true);
```

3. You must extract the additional data by **`get`**( ) methods to expose the values in the receiving application. Because the sample is a text-based display, you can include the **`getString`** ( ) methods in the construction of the string that displays in the console.

   Change:

```
string content = mapMessage.getString("content");
System.Console.WriteLine(sender + ": " + content);
```

   to:

```
string content =
  ("Content: " + mapMessage.getString("content") + "\n" +
   "Distribution: " + mapMessage.getString("Distribution") + "\n" +
   "FiscalYearEnd: " + mapMessage.getString("FiscalYearEnd") + "\n" +
   "LineOfCredit: " + mapMessage.getString("LineOfCredit") + "\n");
System.Console.WriteLine("MapMessage from " + sender +
           "\n------- \n" + content);
```

4. Save and compile the edited `.cs` file.
5. Run the edited `.exe` file.

Now when the `MapTalk` sample runs, the content is the text you typed plus the mapped, resequenced, and converted properties.

Chapter 4 SonicMQ Connections

This chapter explains the programming concepts and actions required to establish and maintain SonicMQ connections. It contains the following sections:

Overview of SonicMQ Connections

The SonicMQ clients provide a lightweight platform to access the messaging features provided by SonicMQ brokers. A client programmer creates connections that establish the application's identity and specify how the connection with the broker is maintained. Within each connection, one or more sessions are established. Each session is used for a unique delivery thread for messages delivered to the client application. This chapter explains the programming required to establish and maintain client connections to brokers. Chapter 5, "SonicMQ Client Sessions," explains the programming required to establish and maintain client sessions.

A SonicMQ application starts by accessing a `ConnectionFactory` and using it to create a **connection** that binds the client to the broker. `ConnectionFactory` objects are created by the client application.

Within a connection, one or more **sessions** can be created. Each session establishes a single-threaded context in which messages can be sent or received. Figure 5 shows a client application where one connection is made through which one session is established. The client application uses programmatic interfaces to the .NET Client API that are executed through the SonicMQ client runtime on the session.

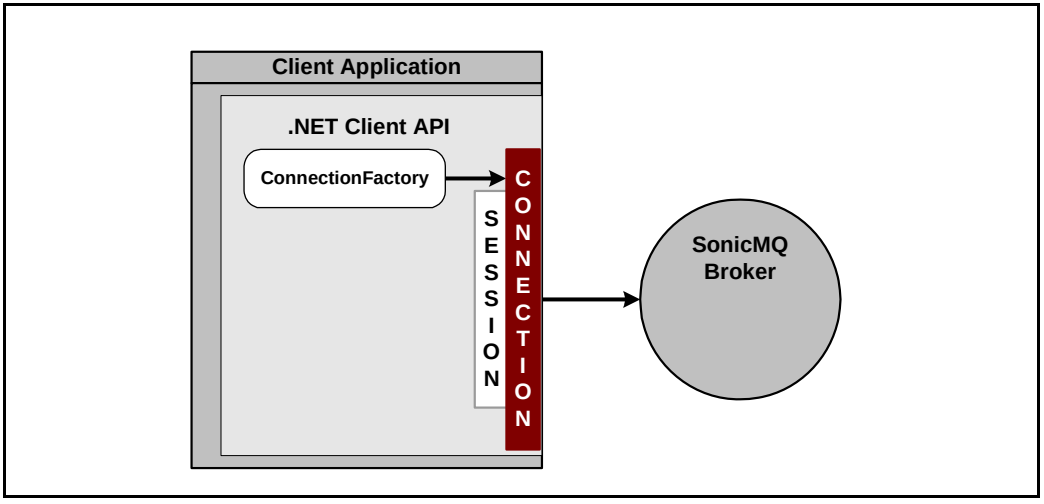


Figure 5. Session on a Connection

Multiple sessions can be established on a single connection. Once the connection and sessions are established, the broker traffic can be either:

- A message producer delivering a message to its broker
- A broker delivering a message to an application that will consume it

In the example shown in Figure 6, two sessions exist on the same connection.

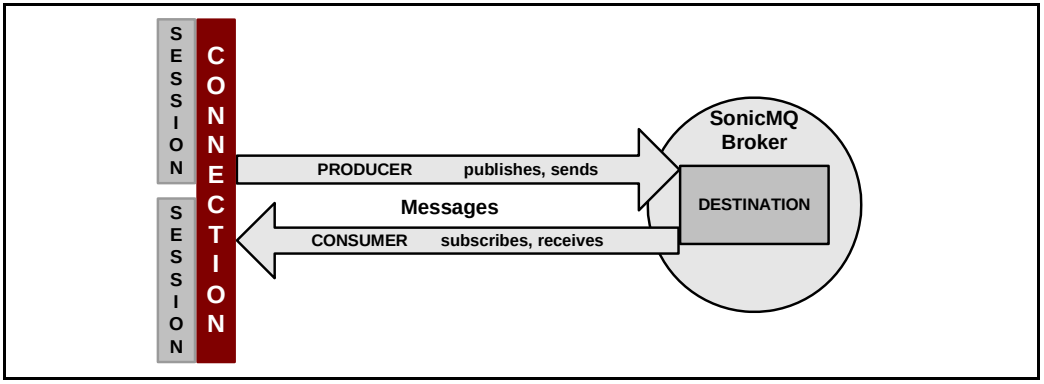


Figure 6. Producers and Consumers

See Chapter 7, "Message Producers and Consumers," for more information about message producers and consumers.

Protocols

This section describes the protocols that .NET client applications can use for broker communication:

- "TCP"
- "SSL"
- "HTTP and HTTPS"

These protocols are nearly transparent within the application. When the port acceptor on the broker matches the connection factory parameter from the application, connection can be established under that protocol.

See "Connecting to SonicMQ Directly" on page 87 for details on explicit use of the protocol value.

TCP

TCP is the default socket type for SonicMQ. Client applications that are Internet-enabled generally use TCP/IP protocols.

SSL

SonicMQ supports encryption at the connection level through SSL. SonicMQ ships with BSAFE-J SSL by RSA Security to ensure secure connections. SonicMQ also supports (but does not include) Java Secure Socket Extensions (JSSE) SSL.

See the *Progress SonicMQ Deployment Guide* for more information about SSL and how to configure SSL on the broker and between brokers.

Before you establish an SSL connection using the SonicMQ .NET Client, you can call the following methods on the `ConnectionFactory`:

- `ConnectionFactory.setSSLCommonName(string` *`commonName`*`)` — Use this method to specify the common name present in the broker certificate for SSL based communication between the client and broker. This name is used by the client to verify the broker certificate. The name specified here is not an X500 string of the form "`CN=[`*`name`*`]`". It just contains the "name" portion of the X500 string. The system performs any formatting if necessary.
- `ConnectionFactory.setSSLCertificate(int` *`certStoreType`*`, string` *`certStoreName`*`, int` *`certSearchPolicy`*`, object` *`certStoreInfo`*`)` — Use this method to specify the certificate to be used by the client in the SSL handshake between the client and broker in case client authentication has been specified on the broker. The system supports multiple certificate store types from where the certificate can be retrieved. This method accepts a certificate store type identifier along with the name of the store and a search policy used to locate the desired certificate from the store. In addition, it accepts a data object that is store type specific and contains information used to load the certificate from the store (for example, private key file names and password).

  This method has no effect for non-SSL broker protocols or client authentication has not been specified at the broker.

HTTP and HTTPS

HTTP is used extensively in SonicMQ. This book focuses on HTTP as a way to establish and maintain client connection to a messaging broker on host port.

Proxies

The SonicMQ .NET client supports proxies for both HTTP and HTTPS. There are two kinds of proxies: transparent proxies and non-transparent proxies (also called tunnels). A transparent proxy does not modify the HTTP request or response beyond what is required for proxy authentication and identification. A non-transparent proxy modifies the HTTP request or response in order to provide some added service, such as group annotation services, media type transformation, protocol reduction, or anonymity filtering.

To enable a proxy for HTTP or HTTPS communication, the SonicMQ .NET client provides the following `ConnectionFactory` method:

```
ConnectionFactory.setHTTPProxyInfo
                        (string proxyHost, int proxyPort, bool proxyTunnel)
```

You must specify proxy information before the connection to the broker is established. Invoking this method subsequent to connection establishment has no effect on the established connection. It takes effect on subsequent connections created by the factory.

This method accepts the proxy host name and port along with a boolean parameter indicating whether you want to tunnel through the proxy or not (that is, treat the proxy as transparent).

The only supported proxy for the SonicMQ .NET client is the Microsoft ISA Server 2000. Consequently, the SonicMQ .NET client supports HTTP only with non-transparent proxies and HTTPS only with transparent proxies. That is, when using `ConnectionFactory.setHTTPProxyInfo( )`with HTTP, you must specify *`proxyTunnel`* as `false`. Also, when using `ConnectionFactory.setHTTPProxyInfo( )` with HTTPS, you must specify*`proxyTunnel`* as `true`. These settings are specific to the Microsoft ISA Server.

Reverse Proxies

A reverse proxy is a front for a Web server that (typically) resides in the DMZ. An application (such as a Web browser or the SonicMQ .NET client) specifies the reverse proxy address, and the reverse proxy is configured to forward the request to an appropriate Web server behind the firewall. The SonicMQ .NET client requires no additional steps to work with reverse proxies. However, the reverse proxy address must be specified in the HTTP/HTTPS URL used for the connection.

HTTPS

If you want to establish HTTPS connections with the SonicMQ .NET client, you must specify SSL settings for the `ConnectionFactory` (see "SSL" on page 78).

HTTP Direct

SonicMQ can interface with pure HTTP Web applications and Web Servers:

- Inbound to the SonicMQ broker, protocol handlers on acceptors let SonicMQ act as a Web Server, transforming received HTTP documents into SonicMQ messages.
- Outbound from the SonicMQ broker, sending messages to routing connections that translate the message into a well-formed HTTP message before sending to the designated URL (typically a Web server).

HTTP Direct is a way to handle messages one-by-one at the broker without establishing connections and sessions. Other than programmatically setting X-HTTP-* properties on the message outbound to the routing node (see page 159 for details), this book does not discuss the general functionality of HTTP Direct. See the *Progress SonicMQ Deployment Guide* section on "Using HTTP(S) Direct" for information about HTTP Direct.

Using HTTP in a connection attempts to use the host and port that you designate as an entry point to HTTP tunneling. See the "TCP and HTTP Tunneling Protocols" chapter of the *Progress SonicMQ Deployment Guide* for information about HTTP tunneling.

HTTP Authentication

HTTP authentication is a challenge-based authentication protocol where a proxy and/or origin server can challenge a client to authenticate itself before processing its request. A challenge can be sent to the client by either the proxy or the origin server (a reverse proxy can challenge the client on behalf of the origin server). There are multiple authentication schemes supported by various servers. The SonicMQ .NET client supports the Basic and NTLM authentication schemes.

The Basic authentication scheme is a simple HTTP authentication scheme where a client sends its username and password (in clear text using base64 encoding) to the server. It was originally defined for HTTP 1.0 and has carried forward to HTTP 1.1. The NTLM authentication schema is a Microsoft-specific authentication scheme.

When challenging a client, a server/proxy sends a challenge per scheme that it accepts. For example, a server might support the Basic, NTLM and Kerberos authentication schemes. When a client sends a request, the server/proxy sends back a challenge for the Basic, NTLM, and Kerberos schemes. It is the client's responsibility to respond to the **most secure** of the challenges that it supports. In this example, if the client only supports the Basic scheme, it responds to the Basic challenge. However, if it also supports the NTLM scheme, it responds to the NTLM challenge instead of Basic, since NTLM is more secure. The server/proxy then processes the challenge response and can either respond with more challenges (NTLM consists of two challenge roundtrips) or accept the request.

When establishing a connection, the SonicMQ .NET client specifies the supported authentication schemes along with the credentials for each of the schemes. You specify this information by calling the following method:

- `ConnectionFactory.setHTTPCredentials (bool proxyAuth, int authScheme, object credentials)`

You can invoke this method multiple times to set the credentials for different supported schemes. Invoking this method after a connection is established has no effect on the established connection. It takes effect only for subsequent connections established by the factory. Currently, the SonicMQ .NET client supports only the Basic and NTLM authentication schemes and considers NTLM to be more secure.

Connection Factories and Connections

The following sections describe how to use connection factories to create connections with a SonicMQ broker.

Connection Factories

To establish a connection with the SonicMQ broker, a C# client uses a `ConnectionFactory` object. The different types of connection factories are:

- `ConnectionFactory`
- `XAConnectionFactory`
- `QueueConnectionFactory`
- `TopicConnectionFactory`

C# clients can obtain a connection factory by instantiating a new connection factory object. They can specify connection information in the object constructor (and possibly customizing further using `set` methods on the factory).

Connection factory objects encapsulate information to connect and configure the client connection. This information might be specified or defaulted to include:

- Host, port, and protocol information
- User, password, and other identity information
- Load balancing, fault-tolerance, selector location, and similar connection or session behavioral settings

The most important connection factory, and hence connection, settings are discussed below. Some of the settings are identifiers that differentiate and distinguish client registrations. These identifiers have specific name restrictions, as shown in Table 2.

Important Table 2, "Restricted Characters for Names," lists characters that are not allowed in SonicMQ. You must not use these restricted characters in your identifier names. See also Appendix A of the *Progress SonicMQ Installation and Upgrade Guide* for a complete reference to the use of characters in SonicMQ names.

Table 2. Restricted Characters for Names

Parameter	*Restricted Characters*
`ClientID`	period (.), asterisk (*), pound (#), slash (/), dollar sign ($)
`ConnectID`	period (.), asterisk (*), pound (#), slash (/), dollar sign ($)
`Durable Subscription`	period (.), dollar sign ($), slash (/), backslash (\). Note that asterisk (*), and pound (#) have wildcard meanings.
`username`	asterisk (*), pound (#), dollar sign ($), slash (/), backslash (\)

Note Although a durable subscription name is not a connection factory setting, it is included in Table 2 for completeness.

URL

The uniform resource locator (URL) identifies the broker where the connection is intended. The URL is in the form:

`[`*`protocol`*`://]`*`hostname`*`[:`*`port`*`]`

where:

- *`protocol`* is the broker's communication protocol (default value: `tcp`).
- *`hostname`* is a networked SonicMQ broker machine.
- *`port`* is the port where the broker is listening. The default port value is **`2506`**.
- For HTTP direct, you can also add a *`url extension`* that determines the parameters and factories.

ConnectID

The `ConnectID` determines whether the broker allows multiple connections to be established using a single `username`/`ConnectID` combination. You control the broker's behavior by calling the `ConnectionFactory.setConnectID(String` *`connectID`*`)` method:

- To allow only one connection, provide a valid *`connectID`*.
- To allow unlimited connections, use `null` as the *`connectID`*.

You can create a valid `ConnectID` by combining the `username` with an additional identifier.

Note See Table 2, "Restricted Characters for Names" on page 83 for a list of restricted characters for ConnectID names.

`ConnectID` can also be passed as an argument to a `ConnectionFactory` object constructor.

Username and Password

The username and password define a principal's identity maintained by the SonicMQ broker's authentication domain to authenticate a user with the SonicMQ broker and the broker's authorization policy to establish permissions and access rights. These parameters are optional. When both parameters are omitted, they both default to `""`, an empty string. When security is not enabled, the username is simply a text label.

A username can be:

- Passed as a parameter to a `ConnectionFactory` constructor
- Passed as a parameter to the `ConnectionFactory.createConnection()` method

Under the SSL protocol, client authentication can be achieved by retrieving the `username` from the client certificate. In that case you simply pass the special-purpose `username` **`AUTHENTICATED`**. The password is ignored.

Note See Table 2, "Restricted Characters for Names" on page 83 for a list of restricted characters for usernames.

ClientID

The `ClientID` is a unique identifier that can avoid conflicts for durable subscriptions when many clients might be using the same username and the same subscription name.

◆ **To set the value of the ClientID, either:**

❖ In the client application, immediately after creating a connection, call the `Connection.setClientID(String` *`clientid`*`)` method.

❖ Set the `ClientID` in the `ConnectionFactory`. To do this, call `ConnectionFactory.setClientID(String` *`clientid`*`)` in the client application.

If you preconfigure the `ClientID`, calling `ConnectionFactory.setClientID(String` *`clientid`*`)` throws an `IllegalStateException`.

See Table 2, "Restricted Characters for Names" on page 83 for a list of restricted characters for `ClientID` names.

Load Balancing

Any broker in a cluster can redirect incoming client connections to another broker in the same cluster for the purpose of load balancing. Load balancing must be configured on the broker. The client must also be configured to indicate that it is willing to have a connect request re-directed to another broker.

See the *Progress SonicMQ Configuration and Management Guide* for information about configuring broker load balancing from the Management Console.

- **To configure the client to allow load-balancing redirects of connect requests:**
  - Call `ConnectionFactory.setLoadBalancing(true)` prior to calling the create connection method.
- **To check the client load-balancing setting:**
  - Call `ConnectionFactory.getLoadBalancing()` to return a boolean indicator of whether load-balancing redirects are allowed by the client.

Alternate Connection Lists

Independent of load balancing, a client can specify a list of broker URLs to which the client can connect. The connection is made to the first available broker on the list. Brokers in the list are tried in random or sequential order.

◆ **To create a connection list programmatically:**

1. Create a comma-separated list of broker URLs. The client attempts to connect to brokers in this list.
2. Call `ConnectionFactory.setConnectionURLs(`*`brokerList`*`)` to point to the text list you created. The client connects to the first available broker on the list.
3. Call `ConnectionFactory.setSequential(`*`boolean`*`)` to set whether to start with the first name in the list (`true`) or a random element (`false`).

Important When a client traverses a connection URL list, the client uses the same userId and password for each broker in the list. If a security exception occurs while the client tries to connect to a broker in the list, the connection fails and the client stops any further traversal of the list.

◆ **To check connection lists:**

1. Call `ConnectionFactory.getConnectionURLs()` to return the broker list.
2. Call `ConnectionFactory.getSequential()` to return the boolean indicator of whether the list is used sequentially or randomly.

Obtaining the Connected Broker URL or Node Name

To get the URL or routing node name of the broker that the client connects to as a result of load balancing or alternate connection lists, call the following methods on the connection object, not the factory object.

- For the connected broker's URL, call the method `getBrokerURL`.
- For the connected broker's routing node name, call the method `getRoutingNodeName`.

Setting Server-based Message Selection

Connections where message selectors are used can receive a large number of messages from the broker and select only a few messages for processing. This condition can be relieved by setting the connection to evaluate messages through a given message selector on the broker and then deliver only the qualified message to the client.

For example, in the `SelectorChat` sample, adding a method call chooses message selection on the server. Notice that it is called after the connection factory is created and before the connection is created:

```
Sonic.Jms.ConnectionFactory factory;
factory = (new Sonic.Jms.Cf.Impl.ConnectionFactory (broker));
factory.setSelectorAtBroker(true);
connect = factory.createConnection (username, password);
```

Choosing where message selectors do their filtering does not effect the messages processed, but might drastically reduce the message traffic at the expense of some additional overhead on the broker. These options can also be set on the factory through the Management Console (see the *Progress SonicMQ Configuration and Management Guide* for information).

Connecting to SonicMQ Directly

An application can use the C# API directly to create a new `ConnectionFactory` object, as shown in Figure 7. This method usually hard-wires many default values into the compiled application. Any overrides to the settings can be read in through a properties file or command-line options when the application is started.

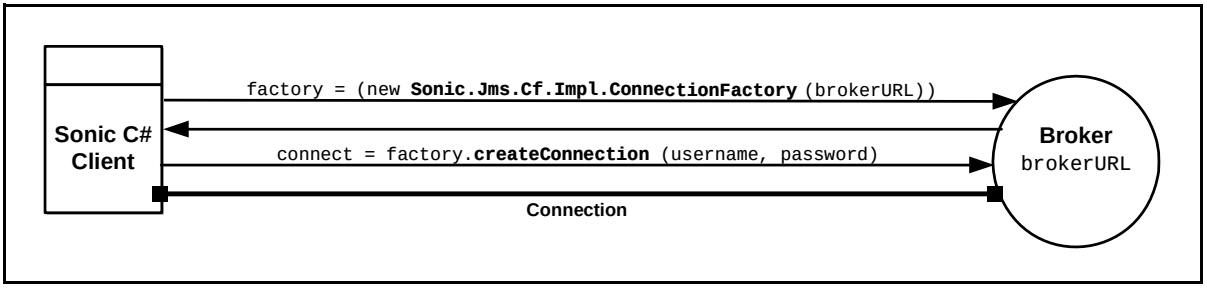


Figure 7. Connecting to SonicMQ Directly

There are several supported constructors for creating a `ConnectionFactory` object. The constructors use combinations of the **brokerURL, brokerHostName, brokerPort, brokerProtocol**, **connectID**, **defaultUsername**, and **defaultPassword** parameters.

Note When user identification is omitted when creating a connection, the connection uses the default values from the `ConnectionFactory`. If authentication is enabled and the username is invalid, a `Sonic.Jms.JMSSecurityException` is thrown.
You can use the common name from a certificate when you use SSL mutual authentication. See the *Progress SonicMQ Deployment Guide* for more information about SSL and security.

Connections

After instantiating a `ConnectionFactory` object, the object's `createConnection( )` methods are used to create a connection. The first action a client must take is to identify and establish a connection with a broker. The following constructors use a connection factory object to get the connection.

Creating a Connection

A `Connection` is an active connection to a SonicMQ broker. A client application uses a connection to create one or more Sessions, threads used for producing and consuming messages.

You create a `Connection` object by using a `ConnectionFactory` object. There are two variants of the `createConnection( )` method:

- Use the default username and password:

```
connect = factory.createConnection( );
```

Important Use this method only when you are not concerned about security.

- Supply the preferred username and its authenticating password:

```
connect = factory.createConnection (username, password);
```

Creating and Monitoring a Connection

Code Sample 1, taken from the `ReliableChat` sample's `setupConnection()` method, shows how to create and monitor a connection. This code uses active pings to check the health of the connection.

Code Sample 1. ReliableChat: setupConnection

```
// Get a connection factory
Sonic.Jms.ConnectionFactory factory = null;
try
{
    factory = (new Sonic.Jms.Cf.Impl.ConnectionFactory (m_broker));
} catch (Sonic.Jms.JMSException jmse) ...

// Wait for a connection.
while (connect == null)
{
    try
    {
        System.Console.Out.WriteLine("Attempting to create connection...");
        connect = (Sonic.Jms.Ext.Connection)
               factory.createConnection(m_username, m_password);
        ...
        // Ping the broker to see if the connection is still active.
        connect.setPingInterval(30);
    }

    catch (Sonic.Jms.JMSException jmse) {
        System.Console.Out.Write("Cannot connect to broker: " + m_broker + ". "
                                                              + jmse.Message);
        System.Console.Out.WriteLine("Pausing " + CONNECTION_RETRY_PERIOD / 1000
                                                 + " seconds before retry.");
        try {
            System.Threading.Thread.Sleep(new System.TimeSpan(10000 *
                                                    CONNECTION_RETRY_PERIOD));
        }
        catch (System.Threading.ThreadInterruptedException) {
        }
        continue;
    }
    ...
```

In Code Sample 1, the `connect.setPingInterval(30)` statement indicates the use of a method that lets the application detect when a connection gets dropped by setting a **`PingInterval`** of 30 seconds. Active pings are a SonicMQ feature that allows an application to check the presence and alertness of the broker on a connection. This technique is particularly useful for connections that listen for messages, but do not send messages.

Invoking **`setPingInterval`**(*`interval_in_seconds`*) on a connection sends a ping message to the broker on that connection at the specified interval to examine the health of the connection.

The broker must respond to each ping sent by the client. If the client does not receive any traffic within a ping interval, the client assumes the connection is bad and drops the connection. See "Handling Dropped Connection Errors" on page 92 for more information. This is true only when the connection is not fault tolerant. When a connection is fault tolerant, pings are still necessary for monitoring the network. When a fault tolerant connection is in use, the ping is activated by default and is set to 30 seconds. However, unlike non-fault tolerant connections, a ping response is not required from the broker, and does not cause a connection drop. See "Fault-Tolerant Connections" on page 93.

You can also configure active pings in the `ConnectionFactory` by invoking the `ConnectionFactory.setPingInterval(`*`interval_in_seconds`*`)` method.

Note Avoid setting a small ping interval. This wastes cycles and your application is burdened with temporary network unavailability. Also, if you set a ping interval that is too small, it might result in false connection drops.

Handling Exceptions on the Connection

The exception handler can handle errors actively. Code Sample 2, from the `ReliableChat` sample, shows how a connection problem initiates a reconnection routine.

Code Sample 2. ReliableChat: Reconnection Routine

```
/// <summary> Handle asynchronous problem with the connection.
/// (as specified in the Sonic.Jms.ExceptionListener interface).
/// </summary>
public virtual void  onException(Sonic.Jms.JMSException jmse)
{
    // See if connection was dropped.

    // Tell the user that there is a problem.
    System.Console.Error.WriteLine("\n\nThere is a problem with the connection.");
    System.Console.Error.WriteLine("   JMSException: " + jmse.Message);

    // See if the error is a dropped connection. If so, try to reconnect.
    // NOTE: the test is against SonicMQ error codes.
    int dropCode = Sonic.Jms.Ext.ErrorCodes.ERR_CONNECTION_DROPPED;
    if (Sonic.Jms.Ext.ErrorCodes.testException(jmse, dropCode)) {
        connectionDropped = true;
        System.Console.Error.WriteLine("Please wait while the application tries to "
                                                    + "re-establish the connection...");
        // Reestablish the connection
        // If we are in connection setup, the setupConnection method itself will retry.
        if (!inSetup) {
            setupConnection();
        }
    }
}
```

Handling Dropped Connection Errors

When broker failure occurs, the existing protocol reset initiates the `onException( )` method of the `ExceptionListener` with the error code: `Sonic.Jms.Ext.ErrorCodes.ERR_CONNECTION_DROPPED`.

In the case of network failure, when a broker becomes disconnected from the network, clients generally do not notice until after they try to publish or send a message. If the client is only subscribing, it might never detect the network failure. By using active ping, the client ensures timely detection of network failures.

Exception Listeners Are Not Intended for Errors

The `ExceptionListener` provides a way to pass information about a problem with a connection by calling the listener's `onException( )` method and passing it a `JMSException` describing the problem.

Using the `ExceptionListener` in this way allows a client to be asynchronously notified of a problem. Some connections only consume messages, and have no other way to learn that their connection has failed. Also, if you have many sessions in the connection, you should not tie reconnect logic to the session. Reconnecting should be done only once at the connection level.

The `ExceptionListener` is not for the purpose of monitoring all exceptions thrown by a connection. The only exceptions delivered to `ExceptionListener` are those that do not have any other place to be reported. If an exception is thrown on a specific method call, the exception must not be delivered to an `ExceptionListener`.

Fault-Tolerant Connections

The client aspect of the Sonic Continuous Availability Architecture is client connections that are fault tolerant. A fault-tolerant connection is designed to be resilient when it detects problems with the broker or network. A standard connection, in contrast, is immediately dropped when the broker or network fails. Because the standard connection is immediately dropped, your client application has to explicitly deal with the situation, possibly trying to create a new connection and resolve any in-doubt messages.

A fault-tolerant connection, unlike a standard connection, is kept alive when the broker or network fails. It automatically performs several tasks on your behalf when a problem occurs. For example, it automatically attempts to reconnect when it encounters a problem with a connection. If it successfully reconnects, it immediately executes several state and sychronization protocol exchanges, allowing it to resychronize client and broker state and resolve in-doubt messages. When the connection successfully resynchronizes client and broker state, the connection is said to be *resumed*, and your client application can continue its operations without any directly visible disruption.

A fault-tolerant connection can respond to broker or network failure in a variety of ways. How it responds depends on how you have deployed SonicMQ and on the nature of the failure:

- If the network experiences a transient failure, the fault-tolerant connection can repeatedly try to recover the connection until the network returns to normal.
- If your client application has redundant network pathways to the broker, one pathway can fail, and the fault-tolerant connection can use the other pathway to resume the connection.
- If your client application is connected to a standalone broker, that fails, the fault-tolerant connection can repeatedly try to reconnect to the broker, until it is recovered and restarted.
- If you configured and deployed a backup broker, and the primary broker fails, the fault-tolerant connection can connect to the backup broker.

Fault-tolerant connections provide continuous operation across failures for operations intended for high reliability:

- Production of `PERSISTENT` messages to both topics and queues.
- Consumption (that is, acknowledgement) of messages from queues and durable subscriptions.
- Production and consumption of messages in a transacted session, integrity of the transaction demarcation operations `commit( )` and `rollback( )`, including duplicate transaction detection.
- Client requests that manage temporary queues.

For more information about fault-tolerant deployments and continuous availability, see the *Progress SonicMQ Deployment Guide*. For more information about configuring fault-tolerant brokers, see the *Progress SonicMQ Configuration and Management Guide*.

Note Fault-tolerant connections are not supported for HTTP Direct.

How Fault-Tolerant Connections are Initially Established

To establish a fault-tolerant connection, two statements are required by the application:

- Request that the connection be fault tolerant which will establish a contract between the broker and the client to perform fault tolerant operations—provided that the broker is licensed to provide continuous availability. The method is:

  `Sonic.Jms.Ext.ConnectionFactory.`**`setFaultTolerant`**`(bool)`

  where `true` requests a fault tolerant connection.
- Supply a connection list with the primary and backup broker connection URLs:

  `Sonic.Jms.Ext.ConnectionFactor.`**`setConnectionURLs`**`(string` *`brokerList`*`)`

  where *`brokerList`* is a comma-delimited list of broker URLs.

ConnectionFactory Methods for Fault-Tolerance

Several `Sonic.Jms.Ext.ConnectionFactory` methods relate to fault tolerant connections:

- Setting and getting `FaultTolerant` described in "Enabling Fault-Tolerant Connections"
- Setting and getting `ConnectionURLs`, described in "Connection Lists" on page 96
- Setting and getting `ClientTransactionBufferSize`, described in "Client Transaction Buffers" on page 97
- Setting and getting `InitialConnectTimeout` and `FaultTolerantReconnectTimeout`, described in "Specifying Connection Timeouts" on page 98

Enabling Fault-Tolerant Connections

By default, a `ConnectionFactory` creates standard (not fault-tolerant) connections. To create a fault-tolerant connection, you must call the following method before you create the connection:

`Sonic.Jms.Ext.ConnectionFactory.`**`setFaultTolerant`**`(Boolean` *`faultTolerant`*`)`

If *`faultTolerant`* is `true`, the `ConnectionFactory` creates fault tolerant connections; if `false`, the `ConnectionFactory` creates standard connections.

To get the `ConnectionFactory`'s current fault-tolerance setting, call the following method:

`Boolean ConnectionFactory.`**`getFaultTolerant`**`( )`

You cannot create a fault-tolerant connection unless the broker is licensed to support-fault tolerance. A broker that is not licensed to support fault tolerance will effectively ignore the `ConnectionFactory` setting. You can determine if a connection is fault tolerant by calling the `Sonic.Jms.Ext.Connection.`**`isFaultTolerant`**`( )` method.

Connection Lists

The client runtime works through a connection URL list, which can include multiple broker URLs. You add URLs to this list by calling the `Sonic.Jms.Ext.ConnectionFactory.`**`setConnectionURLs`**`( )` method.

If you want your client application to use fault-tolerant connections against a replicated broker, the programming model requires you to specify the URLs for your primary and backup brokers in the `ConnectionFactory` URL list. Before the client application initially connects, it does not know which broker (primary or backup) is active. If you omit the active broker from the list, the client cannot initially connect.

See "Alternate Connection Lists" on page 86.

The client runtime works through the list one URL at a time, and connects to the first available broker on the list. The client runtime can work through the list in sequential order, either:

- Starting from a random entry in the list (to get this behavior, you must call the `ConnectionFactory.setSequential(false)` method).

  The following code snippet demonstrates how to make the client runtime randomly choose a broker from a list:

```
//cf is a Sonic.Jms.Ext.ConnectionFactory

cf.setSequential(false);

//Two sets of primary and backup brokers in list
cf.setConnectionURLs("tcp://B1P:2101,tcp://B1B:2102,tcp://B2P:2201,tcp://B2B:2202");
```

- Starting from the beginning of the list (this is the default behavior).

  The following code snippet demonstrates how to make the client runtime choose a broker by starting at the beginning of the list:

```
cf.setSequential(true);

//Two sets of primary and backup brokers in list
cf.setConnectionURLs("tcp://B1P:2101,tcp://B1B:2102,tcp://B2P:2201,tcp://B2B:2202");
```

The following code snippet also demonstrates how to make the client runtime choose a broker by starting at the beginning of the list. However, in this snippet, the primary brokers are listed before their corresponding backup brokers. This approach is appropriate, for example, if the backup brokers are on slower machines than the primary brokers.

```
cf.setSequential(true);

//Two sets of primary and backup brokers in list, primary brokers listed first.
cf.setConnectionURLs("tcp://B1P:2101,tcp://B2P:2201,tcp://B1B:2102,tcp://B2B:2202");
```

When the client runtime successfully establishes a fault tolerant connection with a broker, the broker sends a list of URLs to the client runtime for reconnection. When replicating, the broker also sends a list of standby broker URLs for reconnection.

After a connection is established, you can see the values in these lists by calling the following methods on the `Sonic.Jms.Ext.Connection` object:

- `getBrokerReconnectURLs()`
- `getStandbyBrokerReconnectURLs()`

Client Transaction Buffers

When a fault-tolerant connection fails in the middle of a transaction, the client runtime attempts to resume the connection with the broker. If the broker is down, the client runtime attempts to connect to a standby broker, provided you are using broker replication. If the client runtime can resume the connection with either broker, it must make sure that its transaction state is synchronized with the broker's transaction state.

For performance reasons, the broker buffers transacted messages in memory, instead of saving each message individually as it is received. Consequently, the client runtime also buffers the unsaved messages, so that if the broker goes down and loses the buffered messages, they can be automatically resent by the client runtime.

The broker parameter, **Transactions: Buffer Size** (on the **Tuning** tab of the **Edit Broker Properties** window) specifies the size of the broker's buffer on a per-transaction basis. In general, performance improves as the buffer size is increased. However, the improved performance has two costs—the client runtime uses more memory, and it takes longer to resend the unsaved messages if the broker goes down.

The client application can override the **Transactions: Buffer Size** parameter by calling the following method:

```
ConnectionFactory.setClientTransactionBufferSize(Long size)
```

Valid values for *size* are as follows:

- **Zero (0) (the default)** — Indicates the broker **Transactions: Buffer Size** is applied. The client runtime must be able to buffer up to the broker **Transactions: Buffer Size** parameter per transaction.
- **Positive Long integer** — Specifies the size, in bytes, that the client runtime is willing to buffer per transaction. If the buffer size is reached, client sending threads will block until further messages are saved by the broker. The broker will apply a transaction buffer size that is the lesser of the client-specified value and the broker's **Transactions: Buffer Size**.

  The client runtime must be able to allocate sufficient memory to buffer messages for each active transaction. For local transactions, each Session can have at most one transaction active.

The broker flushes transacted messages to disk when the amount of transacted messages exceeds a calculated amount: the lesser of the broker's **Transactions: Buffer Size** parameter or the fault-tolerant client's transaction buffer size.

To get the client's transaction buffer size, call the following method:

```
Sonic.Jms.Ext.ConnectionFactory.getClientTransactionBufferSize( )
```

Specifying Connection Timeouts

When a client application tries to establish an initial connection or resume a fault-tolerant connection, it might not succeed immediately. The client can continue to try until it succeeds, or it can specify a time interval (timeout) beyond which it will stop trying.

The client application can specify two timeouts related to fault tolerant connections:

- **Initial connect timeout** — How long the client runtime tries to establish an initial connection to the broker.
- **Fault tolerant reconnect timeout** — How long the client runtime tries to resume a fault tolerant connection after a problem is detected.

Initial Connect Timeout

When the client runtime tries to establish an initial connection, it sequentially tries the URLs listed in the `ConnectionFactory`. You can set this list programmatically with the `ConnectionFactory.setConnectionURLs( )` method (see "How Fault-Tolerant Connections are Initially Established" on page 94).

The client runtime continues to try to establish a connection until either a connection is successful or the initial connect timeout is exceeded. If the client runtime is trying to connect to a URL when a timeout occurs, it does not stop immediately. It must complete its current attempt (and fail) before returning a failure to the client application. However, it can return a failure before trying all URLs in the list.

To set the initial connect timeout, call the `setInitialConnectTimeout( )` method:

`Sonic.Jms.Ext.ConnectionFactory.`**`setInitialConnectTimeout`**`(int` *`seconds`*`)`

where *seconds* is one of the following:

- **Positive non-zero value** — Specifies a timeout; the client runtime abandons further connection attempts if the timeout is exceeded.
- **Zero (0)** — Specifies no timeout; the client runtime tries indefinitely.
- **Negative one (-1)** — Specifies that each URL is tried one time only; the client runtime tries each URL sequentially one at a time until a successful connection is made or until all URLs are tried. This sequence is the same as the connection sequence used for standard connections.

If a connection cannot be established within the allocated time, a connection exception is thrown.

You can get the existing initial connect timeout value by calling the following method:

`Sonic.Jms.Ext.ConnectionFactory.`**`getInitialConnectTimeout`**`()`

Fault Tolerant Reconnect Timeout

When a problem is detected with a fault tolerant connection, the client runtime tries to resume the connection. If it can connect to the same broker, it does; otherwise, it tries to reconnect to a standby broker (if you are using broker replication).

You can adjust the timeout or whether the attempts ever timeout by calling the following method:

`Sonic.Jms.Ext.ConnectionFactory.`**`setFaultTolerantReconnectTimeout`**`(int` *`seconds`*`)`

where *seconds* is one of the following:

- **A positive integer** — Specifies a timeout; the client runtime abandons further reconnection attempts if the timeout is exceeded.
- **Zero (0) —** Specifies no timeout; the client runtime tries to reconnect indefinitely.

If the client fails to reconnect in the allocated time, the client is completely disconnected by the broker. A fault-tolerant client runtime that attempts to reconnect late and after the broker has discarded state will encounter a connection failure. When the client runtime fails to resume a connection, the client runtime drops the connection and returns a connection dropped exception to the client application's `ExceptionListener`.

Note The client's ability to reconnect is also influenced by an advanced broker administrative parameter `CONNECTION_TUNING_PARAMETERS.CLIENT_RECONNECT_TIMEOUT`. The default timeout is 10 minutes. By setting this parameter, the administrator can limit the overall length of time the broker maintains state for any fault-tolerant connection that fails and cannot reconnect. The maximum length of time that a broker maintains state is the lesser of the client-specified fault tolerant reconnect timeout and the `CLIENT_RECONNECT_TIMEOUT` setting.

You can get the existing fault-tolerant reconnect timeout value by calling the following method:

`Sonic.Jms.Ext.ConnectionFactory.`**`getFaultTolerantReconnectTimeout`**`()`

Connection Methods for Fault-Tolerance

There are several `Connection` methods related to fault tolerant connections:

- Setting and getting `FaultTolerant` described in "Verifying Whether a Connection Is Fault Tolerant"
- Setting and getting `ConnectionState` and `ConnectionStateListener`, described in "Handling Connection State Changes."
- Setting and getting `BrokerURL`, `BrokerReconnectURLs`, and `StandbyBrokerReconnectURLs`, described in "Getting the URL of the Current Broker and Reconnect Brokers" on page 103

Verifying Whether a Connection Is Fault Tolerant

You can confirm whether a connection you requested as fault tolerant is, in fact, fault tolerant. If the broker is not licensed to enable fault tolerant connections, a standard connection is established. While the `ConnectionFactory.`**`getFaultTolerant`** method might indicate that the application is requesting a fault-tolerant connection, this connection method verifies whether a fault-tolerant connection is active.

`Sonic.Jms.Ext.Connection.`**`isFaultTolerant`**`()`

Handling Connection State Changes

If a fault-tolerant connection fails for some reason, the client runtime reacts differently than it does for a standard connection. When a standard connection fails, the client runtime immediately drops the connection and raises an exception. How the exception is returned to the client application depends on what the application was doing when the exception was raised. If the client application was in the middle of a synchronous call, the exception is thrown by the invoked method. If the exception occurs asynchronously, the client runtime passes an exception to the connection's `ExceptionListener`.

When a fault tolerant connection encounters a problem and cannot communicate with the broker, the client runtime does not immediately drop the connection. Instead, it tries to resume the connection. While trying to resume the connection, it defers passing any exceptions to the client application. If it fails in its attempt to reconnect, it then passes the exceptions to the client application in the same manner as for a standard connection.

While the client runtime is trying to resume a fault-tolerant connection, the client application appears to block. However, the client application can stay informed about the state of the connection by implementing a **`ConnectionStateChangeListener`** and registering it with the appropriate `Connection` object.

Whenever the state of the connection changes, the client runtime calls the listener's **`connectionStateChanged`**`(int state)` method. This method accepts the following valid values (each value represents a different connection state):

- **`Sonic.Jms.Ext.Constants.ACTIVE`** — The connection is active.
- **`Sonic.Jms.Ext.Constants.RECONNECTING`** — The connection is unavailable, but the client runtime is trying to resume the connection.
- **`Sonic.Jms.Ext.Constants.FAILED`** — The client runtime has tried to reconnect and failed.
- **`Sonic.Jms.Ext.Constants.CLOSED`** — The connection is closed.

A client application can obtain the connection's current state by calling the following method on the `Connection` object:

```
int getConnectionState( )
```

If a standard connection calls the `getConnectionState( )` method, it never gets a `RECONNECTING` state.

When a fault tolerant connection is working normally, the connection state is `ACTIVE`. If a problem occurs with the connection, the client runtime changes the state to `RECONNECTING` and attempts to resume the connection. If the attempt is successful, the client runtime changes the state back to `ACTIVE`; if all attempts to reconnect fail, the client runtime changes the state to `FAILED`. Finally, if an `ExceptionListener` is registered, the client runtime calls its `onException( )` method.

When you implement a `ConnectionStateChangeListener`, you must not perform any operations related to the connection, except for calling the following informational methods:

- `Sonic.Jms.Ext.Connection.getConnectionState( )`
- `Sonic.Jms.Ext.Connection.getBrokerURL( )`
- `Sonic.Jms.Ext.Connection.getBrokerReconnectURLs( )`
- `Sonic.Jms.Ext.Connection.getBrokerStandbyReconnectURLs( )`

It is recommended that you do not perform any time- or CPU-intensive processing in the `connectionStateChanged( )` method, as this may impede the client reconnect.

Getting the URL of the Current Broker and Reconnect Brokers

If you are using client URL lists or broker load-balancing, a client connection (fault-tolerant or standard) can be made to one of a number of brokers. Further, with broker load-balancing, it is typical that the URL provided by a load-balancing broker is not configured by the client. With fault-tolerance enabled, the connection can reconnect to a different URL or to a different broker than it initially connected to. In these cases, a client application can determine which broker it is currently connected to by calling the `Sonic.Jms.Ext.Connection.`**`getBrokerURL`**`( )` method. The signature of this method is as follows:

```
public string getBrokerURL( )
```

This method returns the URL of the currently connected broker. If the current connection state is `RECONNECTING`, this method returns the URL of the last broker connected when the connection state was `ACTIVE`. This method can be called after the connection is closed.

URL Lists for Reconnecting

When a client initially establishes a fault-tolerant connection to a broker, the broker passes two URL lists to the client runtime. The first list contains all of the URLs that are tried to resume a connection to the connected broker; the second list contains all of the URLs that are tried to resume a connection to its standby broker.

A client application can access the first list by calling the `Sonic.Jms.Ext.Connection.`**`getBrokerReconnectURLs`**`( )` method. The signature of this method is as follows:

```
public string[] getBrokerReconnectURLs( )
```

A client application can access the second list by calling the `Sonic.Jms.Ext.Connection.`**`getStandbyBrokerReconnectURLs`**`( )` method. The signature of this method is as follows:

```
public String[] getStandbyBrokerReconnectURLs( )
```

Both of these methods are used for purely informational purposes, such as for writing to an audit log. The reconnect logic is automatically performed by the client runtime. These methods can both be called after the fault-tolerant connection is closed.

Broker Reconnect URLs

These are URLs the client runtime can use to try to reconnect to the current broker, in the event of connection failure (transient or other). The broker reconnect URLs allow multiple acceptors on redundant network interfaces to be configured and included in client reconnect logic. The broker reconnect URLs are derived from the configuration by the following rules:

- If the active broker has a default routing URL configured, return the currently connected URL.
- If the active broker has one or more URLs with the same acceptor name as the currently connected URL, return the URLs with same acceptor name, and include the currently connected URL (`getBrokerURL( )`).
- Otherwise return the currently connected URL (`getBrokerURL( )`).

If `getBrokerReconnectURLs( )` is called against a fault-tolerant connection that is `RECONNECTING`, the method returns the broker reconnect URLs when the connection state was last `ACTIVE`.

Standby Broker Reconnect URLs

These are URLs the client runtime can use to connect to a standby broker (a broker that is paired for fault-tolerance with the current broker) if it cannot successfully resume its connection with the current broker. The list of standby broker reconnect URLs is derived from the configuration by the following rules. These rules are consistent with how broker load-balanced connections are selected:

- If the broker is standalone, return null.
- If the standby broker has a default routing URL configured, return the standby broker default routing URL.
- If the standby broker has one or more URLs with same acceptor name as the primary broker URL, return the standby broker URLs with same acceptor name.
- Otherwise return null. The final case is regarded as a configuration error. Replicated brokers must be configured with corresponding acceptor names.

If `getStandbyBrokerReconnectURLs( )` is called against a fault-tolerant connection that is `RECONNECTING`, the method returns the standby broker reconnect URLs of the last broker connected when the connection state was `ACTIVE`.

Reconnect Errors

A fault-tolerant connection might fail to reconnect for a variety of reasons. When a failure occurs, the `ERR_CONNECTION_DROPPED` error code is included in the exception returned to the `Connection's ExceptionListener`. A linked exception provides more information about the specific cause of the failure.

Load Balancing Considerations

When a client is connecting to a replicated broker, both the primary and backup URLs should be specified in the `ConnectionFactory`'s URL list. This holds true if the replicated broker is also a load-balancing broker. If a fault-tolerant client is redirected to a broker that is replicated, the client is automatically capable of reconnecting that broker's list of reconnect URLs and standby reconnect URLs.

To get the URL of the broker that the client connects to as a result of load balancing, call `getBrokerURL( )` on the connection object.

To get the reconnect URLs of the broker that the client connects to as a result of load balancing, call `getBrokerReconnectURLs( )` on the connection object.

To get the URLs of the backup broker for the broker that the client connects to as a result of load balancing, call `getStandbyBrokerReconnectURLs( )` on the connection object.

Acknowledge and Forward Considerations

The acknowledge-and-forward feature allows clients to atomically acknowledge a queue message and move it a new queue. The acknowledge operation and move operation either both succeed or both fail. As part of the acknowledge-and-forward call, the message consumer can optionally change the delivery mode of the message.

The only reliable acknowledge-and-forward operation that is supported with fault-tolerant connections is `PERSISTENT` to `PERSISTENT`.

`PERSISTENT` to `NON_PERSISTENT`, and vice-versa, throws an `IllegalStateException` when attempted on a fault-tolerant connection.

Forward and Reverse Proxies

Fault-tolerant connections work through forward proxy servers. Fault-tolerant connections also work through reverse proxy servers that provide address translation. URLs for primary and backup brokers exterior to the firewall should be configured in the `ConnectionFactory`. When configuring a broker for fault-tolerance behind a firewall, configure the default routing URL to the exterior URL.

Reliability in Fault-Tolerant Connections

This section describes the reliability of various operations in the event of a client reconnect after a failure. In this section, the phrase *broker failure* means one of the following—a broker crashed, recovered fully, and restarted successfully; or a replicated broker crashed and failed over to its backup broker. The general term *failure* means either a broker failure or a transient network failure.

- Production and consumption of persistent messages to temporary queues for fault-tolerant clients are highly reliable across failures.
- Production and consumption of persistent messages to temporary topics are unreliable across failure. For fault tolerant request-reply, applications should use a durable subscriber to handle replies.
- Transaction timeouts (a Sonic-specific feature) are restarted when a broker fails.
- `QueueBrowsers` are unreliable if a fault-tolerant connection detects a problem with the broker or network; all `QueueBrowsers` are immediately closed. The current browse cursor throws a `NoSuchElementException` with text indicating that the browse was terminated due to fail-over. Any attempt to call a `QueueBrowser` method after fail-over results in a `Sonic.Jms.IllegalStateException`. Explanatory text is provided in the exception. The following String error code is provided:

  ```
  Sonic.Jms.Ext.ErrorCodes.BROWSER_CLOSED_DURING_RECONNECT
  ```

- Access to read-exclusive queues might be lost during fail-over. It is possible for a fault-tolerant connection with a `QueueReceiver` open on a read-exclusive queue to fail to reconnect after broker failure. This happens if another client opens a receiver to the same queue before the fault-tolerant client reconnects. In this case, normal connection failure occurs. This problem cannot occur when the client connection recovers from transient network failure.

When a message is sent from a client to an active broker, the client maintains a copy of the message until two acknowledgements are received. The first acknowledgement is from the active broker, the second acknowldegement is from the standby broker through the active broker. If the active broker fails before the second acknowledgement is returned, then—when the client reconnects to the standby as it assumes the active state—it negotiates its state: what was the last message received, what was the last acknowledgement received by the client, and so on. When the now-active broker synchronizes with the client, any missed messages are resent from the client cache. If the active broker fails prior to replication, then—when the client negotiates its state at reconnection—missing messages are resent from the client cache. Messages are removed from the client cache after both acknowledgements are received.

Reconnect Conflict

Connect conflicts are possible during client connection recovery. Conflicts can happen at the connection level and at the durable subscriber level.

Connection Reconnect Conflict

To uniquely identify connections, the `ConnectionFactory` `username` and `connectID` values are used. If `connectID` is `null` (default) a unique connect identifier is allocated by the broker. Therefore, by specifying a null `connectID`, the `username` is permitted to establish any number of connections. If a non-null `connectID` is specified, only one connection with the particular `username` and `connectID` combination is permitted. A client using a non-null `connectID` might be connected when fail-over occurs. Before reconnecting, another client can attempt to connect using the same `connectID` and username identifiers. A normal (non-recovery) connect is received for this client. To the broker this also happens if the original client application disconnects, then attempts to create a new connection. For this reason, it is undesirable for the broker to reject new connects while maintaining state for a fault-tolerant connection that failed and is pending reconnect. Therefore, if a fault-tolerant connection failed and is pending reconnect in the broker and a new connection (non-recovery) is received with the same `connectID` and username, the new connection is accepted and the previous connection state is discarded.

A client using fixed connect identifiers to gain exclusive access can lose such access (have it “stolen” by a different client) during pending reconnect state.

Durable Subscriber Reconnect Conflict

To uniquely identify durable subscriptions, the `ConnectionFactory` `username` and `clientID`, in conjunction with the subscription name parameter provided to `Sonic.Jms.Session.createDurableSubscriber` are used. A client can create a fault-tolerant connection, session, and a durable subscriber. If the connection fails, the connection enters pending reconnect state in the broker. During pending reconnect state, no connection is permitted to create (gain access to) the durable subscriber unless the underlying `connectID` is identical to that of the connection in postponed disconnect state.

Message Reliability

Table 3 describes message reliability levels for clients that reconnect to a broker or its backup after a failure. The reconnect is automatic for fault-tolerant connections, and application driven for standard connections. This table assumes that clients that reconnect using standard connections do not resend in-doubt messages upon reconnecting.

Note The only way to guarantee exactly-once delivery is with a fault-tolerant `MessageProducer` sending messages under `PERSISTENT` delivery mode and a fault-tolerant `MessageConsumer`.

Table 3. Message Reliability

Message Producer		***Message Consumer***			
Connection Type	***Delivery Mode***	***Standard Connection***		***Fault Tolerant Connection***	
		Topic	***Topic (Durable Subscription) or Queue***	***Topic***	***Topic (Durable Subscription) or Queue***
Standard Connection	`DISCARDABLE`	At most once[1]	At most once[1]	At most once[1]	At most once[1]
	`PERSISTENT`	At most once[2]	At least once[1, 2]	At most once[1]	Exactly once[1]
	`NON_PERSISTENT`	At most once[1]	At most once[1]	At most once[1]	At most once[1]
Fault-Tolerant Connection	`DISCARDABLE`	At most once	At most once	At most once	At most once
	`PERSISTENT`	At most once[3]	At least once[2]	At most once	Exactly once
	`NON_PERSISTENT`	At most once	At most once	At most once	At most once

[1]In the case of a standard connection failure, if the last message sent was in doubt, your application logic may decide to retry the publication, after creating a new connection, session, and `MessageProducer`. This causes the generation of a duplicate message if the broker received the original message. This is not considered a redelivery since the message was delivered from a new session. This ambiguity is resolved for fault-tolerant `MessageProducers`: `PERSISTENT` messages are exactly-once; `DISCARDABLE` and `NON_PERSISTENT` messages are dropped in a failure.

[2]In the case of a standard connection failure, the acknowledgement for the last message may be lost. In this case the broker redelivers the message with `JMS_REDELIVERY` set to true.

[3]If a message-consuming client reconnects using a standard connection at the same time as a fault-tolerant publisher is reconnecting, the publisher might resend a message that was delivered to the previously connected client. This is not considered a redelivery since the message was delivered to a new session.

Modifying the Chat Example for Fault-Tolerance

This section describes how to modify the `Chat` sample to use fault-tolerant connections.

◆ To modify the Chat sample:

1. Create the directory *net_client_install_dir*`\samples\csharp\TopicPubSub\`**`ChatFT.`**
2. Create a copy of the file *net_client_install_dir*`\samples\csharp\TopicPubSub\Chat\`**`Chat.cs`** and paste it in the directory you just created.
3. Rename the file **`ChatFT.cs`**.
4. Open the copied file, `ChatFT.cs`, in your C# IDE.
5. Near the top of the file, replace this line:

```
private Sonic.Jms.Connection connect = null;
```

   with this:

```
private Sonic.Jms.Ext.Connection connect = null;
```

6. In the body of the `chatter( )` method, replace the first `try` block with the following:

```
try
{
    Sonic.Jms.ConnectionFactory factory;
    factory = (new Sonic.Jms.Cf.Impl.ConnectionFactory (broker));

    // Tell the ConnectionFactory to create a fault-tolerant connection
    ((Sonic.Jms.Cf.Impl.ConnectionFactory)factory).setFaultTolerant(true);

    // Increase the default connect timeout to 90 seconds
    ((Sonic.Jms.Cf.Impl.ConnectionFactory)factory).
        setInitialConnectTimeout(90);

    // If the connection fails, keep retrying the connection indefinitely
    ((Sonic.Jms.Cf.Impl.ConnectionFactory)factory).
        setFaultTolerantReconnectTimeout(0);

    connect = (Sonic.Jms.Ext.Connection)factory.
        createConnection (username, password);

    // Set the fault-tolerant connection's ConnectionStateChangeListener
    ((Sonic.Jms.Ext.Connection)connect).
        setConnectionStateChangeListener(new ConnectionStateMonitor(connect));

}
```

7. Near the end of the file insert the following class definition:

```
public class ConnectionStateMonitor: Sonic.Jms.Ext.ConnectionStateChangeListener
{
    private Sonic.Jms.Ext.Connection connect = null;

    public ConnectionStateMonitor(Sonic.Jms.Ext.Connection connection)
    {
        connect = connection;
    }
    public void connectionStateChanged(int status)
    {
        System.Console.WriteLine("+++++++++++++++\n");
        // Check status and write appropriate message to the console
        switch (status)
        {
            case Sonic.Jms.Ext.Constants.RECONNECTING:
                System.Console.WriteLine("SYSTEM: Connection is inactive. " +
                    "Trying to reconnect. Please wait."); break;
            case Sonic.Jms.Ext.Constants.ACTIVE:
                System.Console.WriteLine("SYSTEM: Connection is active" +
                    " and operating normally."); break;
            case Sonic.Jms.Ext.Constants.FAILED:
                System.Console.WriteLine("SYSTEM: Connection has failed." +
                    " Cannot reconnect."); break;
            case Sonic.Jms.Ext.Constants.CLOSED:
                System.Console.WriteLine("SYSTEM: Connection is closed."); break;
        }
        // Write the reconnect and standby URLs to the console
        String[] brokerURLs = connect.getBrokerReconnectURLs();
        String[] standbyURLs = connect.getStandbyBrokerReconnectURLs();

        if (brokerURLs == null)
            System.Console.WriteLine("SYSTEM: No broker reconnect URLs provided.");

        if (brokerURLs != null)
        {
            System.Console.WriteLine("SYSTEM: The broker reconnect URLs are as follows:");
            for ( int i = 0; i < brokerURLs.Length; ++i )
            {
                System.Console.WriteLine("Reconnect URL[" + i + "] is " + brokerURLs[i]);
            }
        }

        if ((standbyURLs == null) || (standbyURLs.Length == 0 ))
            System.Console.WriteLine("SYSTEM: No standby broker URLs provided.");

        if (standbyURLs != null && standbyURLs.Length > 0)
        {
            System.Console.WriteLine("SYSTEM: The standby broker URLs are as follows:");
            for ( int i = 0; i < standbyURLs.Length; ++i )
            {
                System.Console.WriteLine("Standby URL[" + i + "] is " + standbyURLs[i]);
            }
        }
    }
}
```

7. Save and compile the modified file.

Running the Modified Chat Example

Now that you modified, saved, and compiled the `Chat` example, you can run through a scenario that demonstrates some of the key differences between fault-tolerant connections and standard connections.

◆ To run the modified Chat example:

1. Make sure the broker is running. If the broker is not running, select:
   Start > Programs > Progress > Sonic 7.6 > Start Domain Manager to start the broker.
2. In a console window at the `ChatFT` directory, enter:
   ```
   Chat -b localhost:2506 -u SALES
   ```
   This step starts a client that uses a fault-tolerant connection.
3. Open another console window to the `Chat` directory and enter:
   ```
   Chat -b localhost:2506 -u MARKETING
   ```
   This step starts a client that uses a standard connection.

   Both clients receive messages posted to the `jms.samples.chat` topic.
4. In the `ChatFT` console window, type some text and press **Enter**.

   The `ChatFT` console window and the `Chat` console window both display the text you entered, preceded by:
   ```
   SALES:
   ```
5. In the `ChatFT` console window, type some text and press **Enter**.

   The `Chat` console window and the `ChatFT` console window both display the text you entered, preceded by:
   ```
   MARKETING:
   ```
6. In the `SonicMQ Container1` console window (in which the broker is running), enter **Ctrl-C.**

   This causes the broker to shut down and close all active connections. You are prompted whether you want to terminate the batch job.

7. In the `SonicMQ Container1` console window, enter **Y** to terminate the batch job.

   The following output is displayed in the `ChatFT` console window:

```
++++++++++++++++

SYSTEM: Connection is inactive. Trying to reconnect. Please wait.
SYSTEM: The broker reconnect URLs are as follows:
Reconnect URL[0] is tcp://localhost:2506
SYSTEM: No standby broker URLs provided.
```

   This output is displayed because the client runtime calls the `connectionStateChanged(int state)` method when it detects a change in the state of the connection. Because this example sets the fault tolerant reconnect timeout to try indefinitely, this client continues to try to reconnect until you explicitly shut down the client or until the connection is resumed. Because the default broker is not configured with a backup broker, no standby broker URLs are listed.

8. In the `Chat` console window, type some text and press **Enter**.

   A message is displayed indicating that the session was closed (which occurred when the standard connection to the broker was closed).

9. In the `ChatFT` console window, type some text and press **Enter**.

   Observe that the client application appears to block. This behavior occurs because all client operations are suspended when the connection is unavailable.

10. Restart the broker by selecting:
    Start > Programs > Progress > Sonic 7.6 > Start Container1.

    When the broker is restarted, the fault-tolerant connection is resumed. This causes the client runtime to call the `connectionStateChanged(int state)` method again, resulting in the following output:

```
++++++++++++++++

SYSTEM: Connection is active and operating normally.
SYSTEM: The broker reconnect URLs are as follows:
Reconnect URL[0] is tcp://localhost:2506
SYSTEM: No standby broker URLs provided.
```

    The `ChatFT` console window also displays the text you entered while the connection was unavailable, preceded by:

    `SALES:`

You completed this example. You can experiment further, or you can close the `ChatFT` and `Chat` console windows.

Starting, Stopping, and Closing Connections

Connections require an explicit **`start`** command to begin the delivery of messages. All sessions within a connection respond concurrently to the connection **`start`**, **`stop`**, and **`close`** events. You do not have to stop or start connections to publish or send messages.

Starting a Connection

To start delivery of incoming messages through a connection, use the `connect.start( )` method. If you stop delivery, messages are still saved for the connection. Under a restart, delivery begins with the oldest unacknowledged message. Starting an already started session is ignored. Use the following syntax to start delivery through a connection:

```
connect.start( )
```

Stopping a Connection

To stop delivery of incoming messages through a connection, use the `connect.stop( )` method. After stopping, no messages are delivered to any message consumers under that connection. If synchronous receivers are used, they block. A stopped connection can still send or publish messages. Stopping an already stopped session is ignored. Use the following syntax to stop delivery through a connection:

```
connect.stop( )
```

When a connection is stopped, that connection is, in effect, paused. The message producers continue to perform their functions. The consumers, however, are not active until the connection restarts. When the `stop( )` method is called, the stop waits until all the message listeners return before it returns. Active message consumers can receive null messages if they are using `receive(timeout)` or `receiveNoWait( )`.

Closing a Connection

To close a connection, use the `connect.close( )` method.

When a connection is closed, all message processing within the connection's one or more sessions is terminated. If a message is available at the time of the close, the message (or a null) can be returned, but the message consumer might receive exceptions by trying to use facilities within the closed connection.

When a transacted session is closed, the transaction in progress is marked for rollback. This is true whether the shutdown was orderly or unplanned, such as a broker or network failure.

The message objects can be used in a closed connection with the exception of the message's acknowledge methods.

See Chapter 5, "SonicMQ Client Sessions," for information about coding connections and sessions and handling exceptions on connections.

Using Multiple Connections

Sometimes it can be advantageous to use multiple connections in an application, even though the ordering of messages is only assured within a session (a single thread of execution). The sheer volume of information flowing through the connection might warrant multiple connections rather than multiple sessions. Figure 8 shows two connections to a SonicMQ broker, each with two sessions.

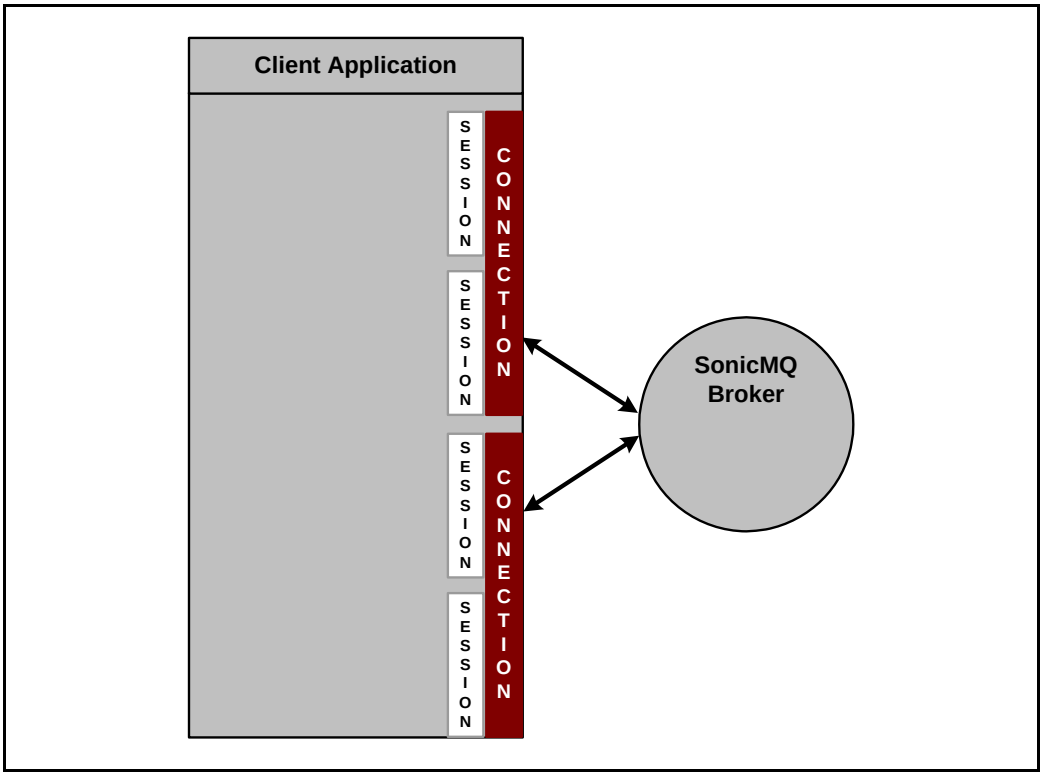


Figure 8. Multiple Connections in a Client Application

Communication Layer

The SonicMQ broker works in concert with the network layer to provide asynchronous message communications between client applications. As shown in Figure 9, a client can send and receive messages through the SonicMQ API and interfaces to communicate on network connection to a broker. Messages might be stored in a message store as an optional service specified by the message producer.

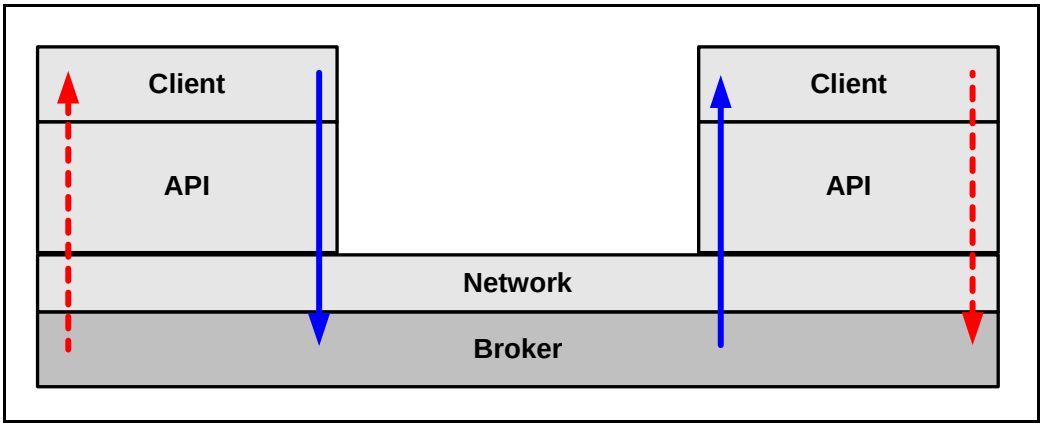


Figure 9. Client-Broker-Client Communications

The connection layer, as shown in Figure 10, involves getting a `ConnectionFactory`, then creating a `Connection`, and finally creating a `Session`. A `Session` holds `MessageProducer` and `MessageConsumer` objects.

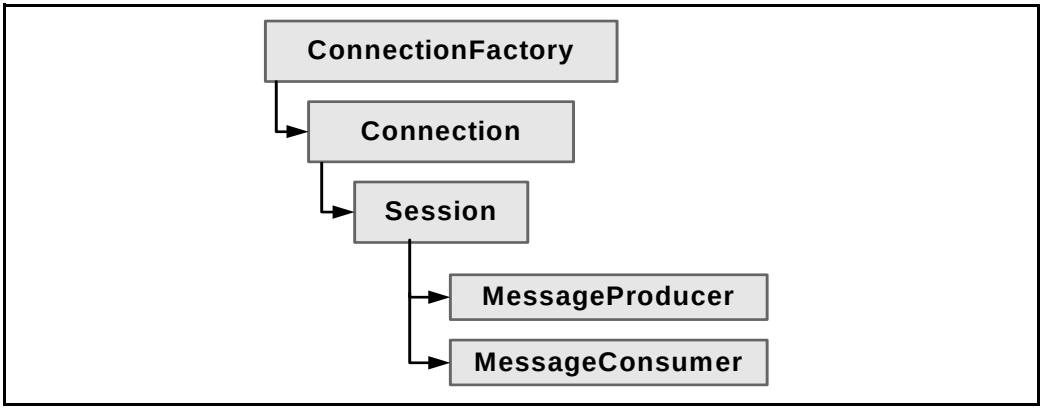


Figure 10. Sessions in Connections from Connection Factories

Each instance of a `MessageConsumer` is dedicated to only one of the messaging models:

- **Point-to-point (PTP)** — Messaging is *one-to-one* because only one consumer receives the message. Messages are placed on queues where they endure until a consumer takes delivery and acknowledges receipt.
- **Publish and Subscribe (Pub/Sub)** — Messaging is *one-to-many* or *broadcast* because there can be any number (between zero and many) of consumers for a given topic who each receive the one message that was sent. In addition, a consumer can be a durable subscriber, and SonicMQ saves messages until the subscriber reconnects. If no consumers express an interest in a message topic, the message is discarded.

Chapter 5 SonicMQ Client Sessions

This chapter describes SonicMQ client sessions; it contains the following sections:

Overview of Client Sessions

A `Session` object represents a client **session**: a single-threaded context for producing and consuming messages. A `Session` object serves as a factory for `MessageConsumer` and `MessageProducer` objects. These objects, once created, are associated with the `Session` object throughout its lifetime.

To create a `Session` object, you use a `Connection` object, which provides a `createSession( )` method. This method can be called multiple times to create multiple `Session` objects. These objects, once created, are associated with the `Connection` object throughout its lifetime. The signature of the `createSession()` method is:

```
Sonic.Jms.Session createSession( Boolean transacted, int acknowledgementMode )
```

where:

- *`transacted`* — [`true | false`]

  If `true`, the session is transacted and the *`acknowledgementMode`* setting has no effect, because the transaction implicitly handles acknowledgement.

- *`acknowledgementMode`* —

  ```
  [ Sonic.Jms.SessionMode.AUTO_ACKNOWLEDGE |
    Sonic.Jms.SessionMode.CLIENT_ACKNOWLEDGE |
    Sonic.Jms.SessionMode.DUPS_OK_ACKNOWLEDGE |
    Sonic.Jms.Ext.SessionMode.SINGLE_MESSAGE_ACKNOWLEDGE ]
  ```

  Indicates whether the client acknowledges messages it receives. If the session only produces messages, *`acknowledgementMode`* has no effect

Acknowledgement Mode

A client application, when it receives a message from the broker, must acknowledge receipt of the message. How it acknowledges the message depends on the `Session` object's acknowledgement mode.

SonicMQ supports four acknowledgement modes, described below. Two of these modes require explicit acknowledgements; two require implicit acknowledgements. An explicit acknowledgement requires the client application to call the `acknowledge( )` method, which is defined as part of the `Sonic.Jms.Message` interface. An implicit acknowledgement, in contrast, requires the `Session` object to automatically acknowledge messages on behalf of the client application.

When a client application creates the `Session` object, it sets the object's acknowledgement mode (the setting holds throughout the `Session` object's lifetime). The four acknowledgement modes are:

- **`AUTO_ACKNOWLEDGE`** (implicit) — The `Session` object automatically acknowledges receipt of each message. After a system failure, the last message might be redelivered.
- **`DUPS_OK_ACKNOWLEDGE`** (implicit) — The `Session` object "lazily" acknowledges messages. After a system failure, multiple messages might be redelivered.
- **`CLIENT_ACKNOWLEDGE`** (explicit)— The `acknowledge( )` method, when called by the client application, acknowledges the receipt of all messages received by the session. After a system failure, all unacknowledged messages might be redelivered.
- **`SINGLE_MESSAGE_ACKNOWLEDGE`** (explicit) — The `acknowledge( )` method, when called by the client application, acknowledges the receipt of only the current message and no preceding messages. After a system failure, all unacknowledged messages might be redelivered.

These modes determine how the client application sends acknowledgements to the broker. They do **not** determine how the message consuming application interacts with the application that produced the message.

A message-producing application, if it wants a reply to its message, can set the message's `JMSReplyTo` header. It can also set the message's `JMSCorrelationID` header field, if it wants to match the reply with a particular request.

Recover

A client application might build up a large number of unacknowledged messages while attempting to process them. A `Session` object's `recover( )` method is used to stop a session and restart it with its first unacknowledged message.

A `recover( )` action notification tells SonicMQ to stop message delivery in the session, set the `redelivered` flag on unacknowledged messages it will redeliver under the recovery, and then resume delivery of messages, possibly in a different order than originally delivered.

The `recover( )` method is most important when the acknowledgement mode is `CLIENT_ACKNOWLEDGE` or `SINGLE_MESSAGE_ACKNOWLEDGE`.

Explicit Acknowledgement

A client application can explicitly acknowledge a message by calling the `acknowledge( )` method, which is defined as part of the `Sonic.Jms.Message` interface. The behavior of this method depends on the `Session` object's acknowledgement mode:

- **`AUTO_ACKNOWLEDGE`** — The method is ignored.
- **`CLIENT_ACKNOWLEDGE`** — The method explicitly acknowledges all unacknowledged messages received by the session.
- **`DUPS_OK_ACKNOWLEDGE`** — The method is ignored.
- **`SINGLE_MESSAGE_ACKNOWLEDGE`** — The method explicitly acknowledges the current message.

If the `Session` is transacted, the method is ignored.

Transacted Sessions

When a `Session` is **transacted**, it combines a group of one or more messages with client-to-broker ACID properties: Atomic, Consistent, Isolated, and Durable. Message input and message output are staged on the broker system but not completed until you call the method to complete the transaction. Completion of a transaction, determined by your code, does one of the following:

- **Commit** — The series of messages is sent to consumers.
- **RollBack** — The series of messages (if any) is destroyed.

The completion of a `Session`'s current transaction automatically begins the next transaction. A transacted `Session` impacts producers and consumers, as described in Table 4.

Table 4. Transacted Session Events by Message Role

Role	`commit( )`	`rollback( )`
Producer	Delivers the series of messages staged since the last call.	Disposes of the series of produced messages staged since the last call.
Consumer	Acknowledges the series of messages received since the last call.	Redelivers the series of received messages retained since the last call.

When a rollback is performed in a session that is both sending and receiving, its produced messages are destroyed and its consumed messages are automatically recovered.

Rollbacks can be either explicit or implicit. Explicit rollbacks occur when the client calls the `rollback( )` method. Implicit rollbacks occur when either:

- The session or connection is closed without finishing the transaction
- The application, connection, or broker experiences failure

To check whether a session is transacted, use the `getTransacted( )` method. The return value is `true` if the session is in transacted mode.

A transacted session only completes successfully when an explicit `commit( )` is invoked.

Broker-managed Timeouts on Transacted Sessions

An undetected hang occurring in a session can lead to unexpected behavior, because a message staged as part of a transaction is indefinitely invisible. For message consumers reading from a queue, the message is neither committed so that it can be further processed nor released so that it can be put back on its queue. For message producers, the message is not accessible to a consumer so that it might be processed further.

A SonicMQ broker can use a broker configuration property to indicate that it does not tolerate transacted messages in process more than the specified number of minutes. If the time is exceeded, the transaction is forced to roll back and the transacted session is then closed. This property is the transaction **Idle Timeout** property. You can configure this property in the Sonic Management Console by selecting **Broker Properties** and then selecting the **Tuning** tab. The **Idle Timeout** property is in the **Transaction** section.

The timeout interval can be **0** (to never idle-out a transaction in process) or any positive integer value that represents the number of minutes of inactivity before the broker managed timeout is enforced. As you have no way of knowing the broker's rules, you should try to complete transactions as soon as possible.

Without broker-managed timeouts, the transaction rolls back when the application disconnects or shuts down.

Duplicate Message Detection

The broker can be set up to commit transactions, so that they index a universally unique, 32-character identifier (UUID) supplied by the sender. You should make this UUID a meaningful name within your application, for example, `order number`, `customer number`, `authorization number`, etc. The sender then uses a commit method to commit the transacted messages (unless a transaction identifier previously sent is still unexpired). Otherwise a rollback of the transaction is forced and a `Sonic.Jms.TransactionalRolledBackException` is thrown by the `commit( )` method.

The signature of this type of commit is:

```
Session.commit(String transactionID, long timeToLive)
```

where *`transactionID`* is a UUID and *`timeToLive`* is the intended lifespan of the indexed identifier in milliseconds. If you omit the *`timeToLive,`* the target broker's advanced property `DUPLICATE_DETECTION_PARAMETERS.INDEXED_TXN_DEFAULT_LIFESPAN` sets the lifespan of the indexed identifier. You can configure advanced properties in the Sonic Management Console by selecting **Broker Properties** and, on the **Advanced** tab, selecting **Edit Advanced Properties**.

Alternatively, for the *transactionID*, you can use a hashcode calculated over the message payload instead of a UUID. The hashcode must be unique for each transaction being tracked within the transaction age limit (*timeToLive*).

See "Duplicate Message Detection Overview" on page 254 for more information about detecting duplicate messages.

Session Objects

`Session` objects allow creation of the destinations, producers, consumers, and messages that are used in the session, as shown in Figure 11.

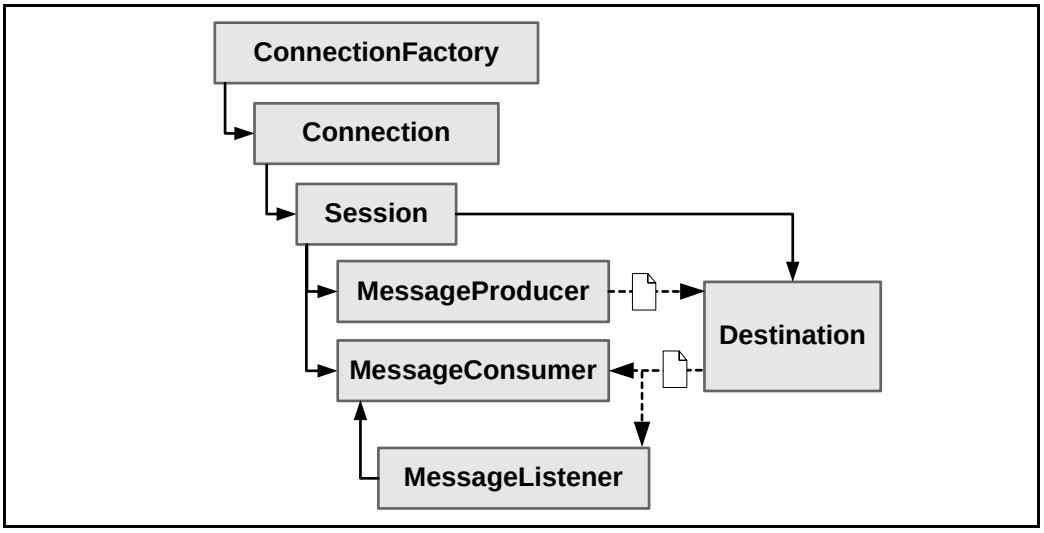


Figure 11. Primary Session Objects

Figure 12 shows the types of message objects that are created from session methods.

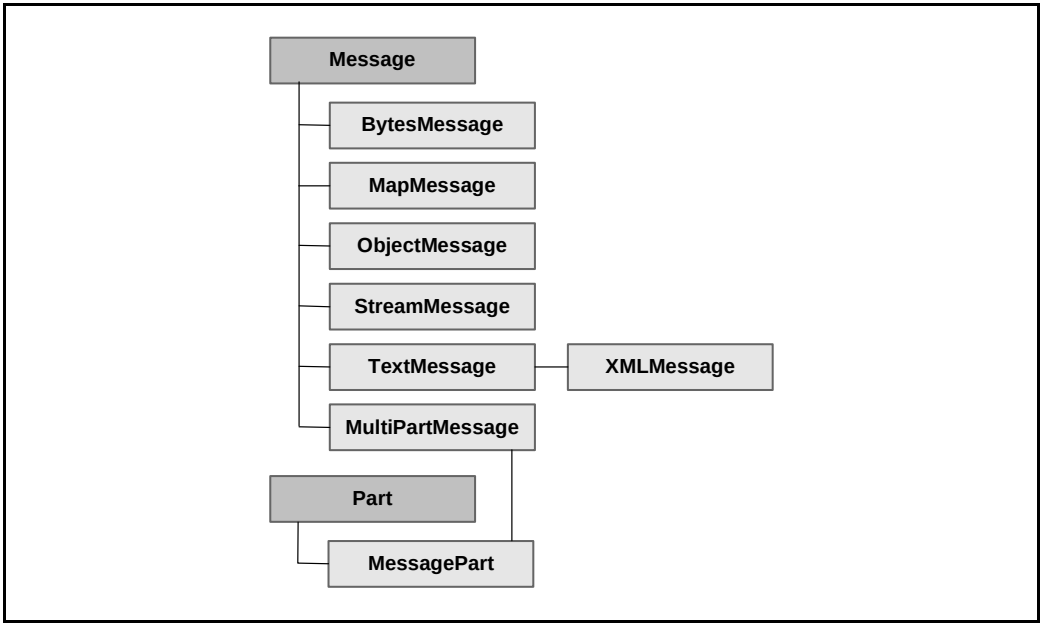


Figure 12. Types of SonicMQ Message Objects

The `Sonic.Jms.Ext.XMLMessage` type is unique to SonicMQ and extends `TextMessage`; the `Sonic.Jms.Ext.MultipartMessage` type is unique to SonicMQ and extends `Message`.

Creating a Destination

Destinations are named locations where messages are stored. In SonicMQ, destinations are queues or topics.

Point-to-point: createQueue

To create a queue, call the following method:

```
Sonic.Jms.Queue queue = session.createQueue(String queueName)
```

where *queueName* is the name of a queue that currently exists on a broker. If security is defined for queues, the client application might be unable to read from or write to a queue.

Publish and Subscribe: createTopic

To create a topic, call the following method:

```
Sonic.Jms.Topic topic = session.createTopic(String topicName)
```

where *topicName* is an arbitrary topic name of up to 256 Unicode characters (the name cannot begin with "SonicMQ"). If security is defined for topics, the client application might be unable to read from or write to a topic.

See Chapter 11, "Hierarchical Name Spaces," for topic name restrictions.

Temporary Destinations

Temporary destinations (**TemporaryTopic** or **TemporaryQueue**) can be created for request-and-reply mechanisms. See "Reply-to Mechanisms" on page 188 for more information.

Creating a MessageProducer

A `MessageProducer` sends messages to one or more destinations.

You create a `MessageProducer` object by calling a `Session` object's `createProducer( )` method:

```
public Sonic.Jms.MessageProducer
                    createProducer(Sonic.Jms.Destination destination)
```

Since `Queue` and `Topic` extend `Destination`, they are valid parameters. If you provide a `Destination`, the returned `MessageProducer` uses the `Destination` as its default. If you use `null` as the `Destination`, the returned `MessageProducer` is not tied to any particular `Destination`.

Creating a MessageConsumer

A `MessageConsumer` receives messages from a single `destination`.

You create a `MessageConsumer` object by calling one of the `Session` object's `createConsumer( )` methods:

- ```
  public Sonic.Jms.MessageConsumer
       createConsumer(Sonic.Jms.Destination destination)
  ```
- ```
  public MessageConsumer
       createConsumer(Sonic.Jms.Destination destination,
                      String messageSelector)
  ```
- ```
  public MessageConsumer
       createConsumer(Sonic.Jms.Destination destination,
                      String messageSelector,
                      Boolean NoLocal)
  ```

Since both `Queue` and `Topic` extend `Destination`, either is a valid `Destination`.

The returned `MessageConsumer` object is dedicated to the `Destination` you provide. If dedicated to a `Queue`, it follows PTP messaging model; if to a `Topic`, the Pub/Sub messaging model.

To create a `MessageConsumer` that is a durable subscriber to a `Topic`, call one of the `Session` object's `createDurableSubscriber( )` methods:

- ```
  public Sonic.Jms.TopicSubscriber
       createDurableSubscriber(Sonic.Jms.Topic topic, String name)
  ```
- ```
  public Sonic.Jms.TopicSubscriber
       createDurableSubscriber(Sonic.Jms.Topic topic,
                               String name)
                               String messageSelector,
                                 Boolean NoLocal)
  ```
  ```

Creating a Message

You create a message by calling the appropriate method on the `Session`. The method names follow this general pattern:

```
Sonic.Jms.[type]Message msg = sendSession.create[type]Message( )
```

where *type* is the message type:

- `Sonic.Jms.TextMessage msg = sendSession.createTextMessage( )`
- `Sonic.Jms.BytesMessage msg = sendSession.createBytesMessage( )`
- `Sonic.Jms.MapMessage msg = sendSession.createMapMessage( )`
- `Sonic.Jms.Message msg = sendSession.createMessage( )`
- `Sonic.Jms.ObjectMessage msg = sendSession.createObjectMessage( )`
- `Sonic.Jms.StreamMessage msg = sendSession.createStreamMessage( )`

To create an **`XMLMessage`** or **`MultipartMessage`**, you use methods defined for the `Sonic.Jms.Ext.Session` interface (which extends `Sonic.Jms.Session`):

- ```
  Sonic.Jms.Ext.XMLMessage msg =
      (Sonic.Jms.Ext.Session)sendSession.createXMLMessage( )
  ```
- ```
  Sonic.Jms.Ext.MultipartMessage msg =
      (Sonic.Jms.Ext.Session)sendSession.createMultipartMessage( )
  ```

See Chapter 6, “Messages,” for information about message interfaces, structure, and fields.

Closing a Session

Each session should only have a single thread of execution. The `close( )` method is the only `Session` method that can be called while another session method is executing in another thread.

Closing a `CLIENT_ACKNOWLEDGE` session does not force an `acknowledge( )` to occur. Attempts to use a closed connection's session objects causes an `IllegalStateException`. Starting a started connection or closing a closed connection has no effect and does not throw an exception.

The `Message` objects can be used in a closed session (with the exception of the message's `acknowledge( )` method).

When the connection closes, its sessions are implicitly closed.

Flow Control

The asynchronous benefits of SonicMQ are not limited to simply receiving without blocking. They also include:

- Send and receive buffers that stage messages in transit between a client application and a broker
- An optimized persistence mechanism to maximize broker performance for guaranteed message delivery
- Concurrent Transacted Cache technology that uses in-memory cache and high-speed log files to increase throughput for short-duration persistent messages
- Queues defined with specified amounts of memory and disk space reserved for the queue content

Any of these resources might be offered more data than can be managed. If flow control is active, SonicMQ throttles back the message flow from the producer, allowing the next message to flow into the buffers only when space is available.

In Pub/Sub and PTP you can disable flow control so when resources are nearly exhausted, SonicMQ can, under programmatic control, generate exceptions until flow control conditions are cleared.

When flow control is active, the messages might be sent to consumers at a rate faster than the messages are actually consumed. When the buffers that store unprocessed messages approach the flow control threshold, flow control can stop new additions until the buffers fall below a threshold level.

The back pressure from slower consumption might start to impact the buffers for queues or durable subscriptions. When system or queue capacities are filled with messages in process, flow control is activated against producers. The message acceptance rate drops, which eventually results in back pressure at the producers, causing them to either tolerate the slowdowns or, with flow control disabled, to throw an exception so that you can handle the situation. For example, you can catch the exception and have the application wait some period of time before resending.

To avoid the invocation of flow control you can:

- Optimize application processing on incoming messages.
- Adjust the consumer buffer (on the broker side).
- Increase the size of queues.
- Decrease the message expiration time of messages.
- Use `DISCARDABLE`.

Note Messages sent to a queue only expire after they are placed on the queue, so expiration detection can only result from:

- Dequeue operations by receivers
- Processing by the queue cleanup thread

Browsing the queue does not detect expiration.

Flow Control Management Notifications

SonicMQ can provide administrative notification when flow control is preventing a `MessageProducer` from producing messages over a significant period of time. These notifications contain information that identifies problems—such as a very slow subscriber or a queue that is not being serviced by receivers—so corrective action can be taken.

Flow control is triggered on a regular basis when a broker is under load, perhaps several times every second. These intermittent conditions are usually transient and unremarkable. However, when flow control blocking is sustained, an application producer session can be prevented from producing messages for a significant period of time.

Monitoring Intervals

The monitoring interval is a property of the `ConnectionFactory` that is set before connections are created. You set the monitoring level by calling `ConnectionFactory.`**`setMonitorInterval`**`(Int32 `*`interval`*`)`.

The value found in the factory when a connection is created applies to any sessions created by that connection, and cannot be subsequently modified. The property defines the duration of the monitoring interval in seconds, where **`0`** indicates that flow control monitoring is disabled for all sessions on the connection.

Since flow control pause notifications are generated after the session is blocked for one full monitoring interval, it might take as long as another monitoring interval from the time the session becomes blocked before a notification is generated.

The block-detection logic monitors whether one or more produced messages remain blocked in the client buffers due to flow control. The logic does not monitor conditions where the client is unable to send a message due to network congestion or other load-related factors.

If a producer session remains blocked over multiple monitoring intervals, a flow control pause notification is generated at the end of each monitoring interval as long as the producer session remains blocked. When the session becomes unblocked, a flow control resume notification is generated.

Pub/Sub

In Pub/Sub messaging, when a block is sustained throughout a monitoring interval, an administrative notification is generated that identifies:

- `Username` and `ConnectID` of the blocked producer session
- `Username`, `ConnectID`, and `Topic` of any non-durable subscriber that is blocking the producer session
- `Username`, `ClientID`, and subscriber name of any durable subscriber that is blocking the producer session

When the block is relieved, another administrative notification is generated identifying the `Username` and `ConnectID` of the now-unblocked producer session.

PTP

In PTP messaging, when a block is sustained throughout a monitoring interval, an administrative notification is generated that identifies:

- `Username` and `ConnectID` of the blocked producer session
- Name of queue that is blocking the producer session or routing queue

When the block is relieved, another administrative notification is generated identifying the `Username` and `ConnectID` of the now-unblocked producer session

Disabling Flow Control

You can disable flow control so that applications can catch the exceptions thrown when messages sent cause flow problems on the broker. To disable flow control, call the `Session.`**`setFlowControlDisabled`**`(Boolean` *`disabled`*`)` method.

where **`TRUE`** indicates that flow control is not active in the session.

Flow to Disk

If flow control is active, `MessageProducers` might block, waiting for `MessageConsumers` to process messages accumulated in in-memory buffers. The flow-to-disk feature relieves this problem by temporarily writing messages to disk, allowing message production to continue despite slow message consumption. This feature is designed for Pub/Sub messaging, in which one slow consumer might hold up message production for other consumers.

For a detailed description of flow-to-disk functionality, see the *Progress SonicMQ Performance Tuning Guide*.

An administrator can enable this feature for all clients connected to a broker by setting a broker configuration parameter (`FLOW_TO_DISK`). You can programmatically override the broker setting for all sessions by calling the following method:

`Sonic.Jms.Ext.ConnectionFactory.`**`setFlowToDisk`**`(Integer` *`flowSetting`*`)`

where *`flowSetting`* is an `Integer` set to one of the following values:

- **`Sonic.Jms.Ext.Constants.FLOW_TO_DISK_USE_BROKER_SETTING`** (the default) — Specifies that the broker setting of `FLOW_TO_DISK` is used for the consumer.
- **`Sonic.Jms.Ext.Constants.FLOW_TO_DISK_ON`** — Specifies that `FLOW_TO_DISK` is on for the consumer, regardless of the broker setting.
- **`Sonic.Jms.Ext.Constants.FLOW_TO_DISK_OFF`** — Specifies that `FLOW_TO_DISK` is off for the consumer, regardless of the broker setting.

To override this setting for a single `Session`, call the following method:

`Sonic.Jms.Ext.Session.`**`setFlowToDisk`**`(Int32` *`flowSetting`*`)`

where the allowable values for *`flowSetting`* are the same as for the `ConnectionFactory.setFlowToDisk( )` method, except that parameters are passed as `ints`, not `Integers`.

Only a subscriber can meaningfully set the `FLOW_TO_DISK` setting. If a session exclusively produces messages, calling the `Sonic.Jms.Ext.Session.setFlowToDisk( )` method has no effect.

Using Sessions and Consumers

There are many advantages to using multiple connections and multiple sessions in an application even though the ordering of messages is only assured within a session (a single thread of execution).

Multiple Sessions on a Connection

If you use multiple sessions, you cannot take advantage of serialized operations on a single thread of execution. Multiple sessions are best suited for alternate or supporting functions within an application. Figure 13 shows multiple sessions using two sessions and only one connection. As the connection is associated with a messaging domain—PTP or Pub/Sub—multiple sessions are constrained to the connection's domain.

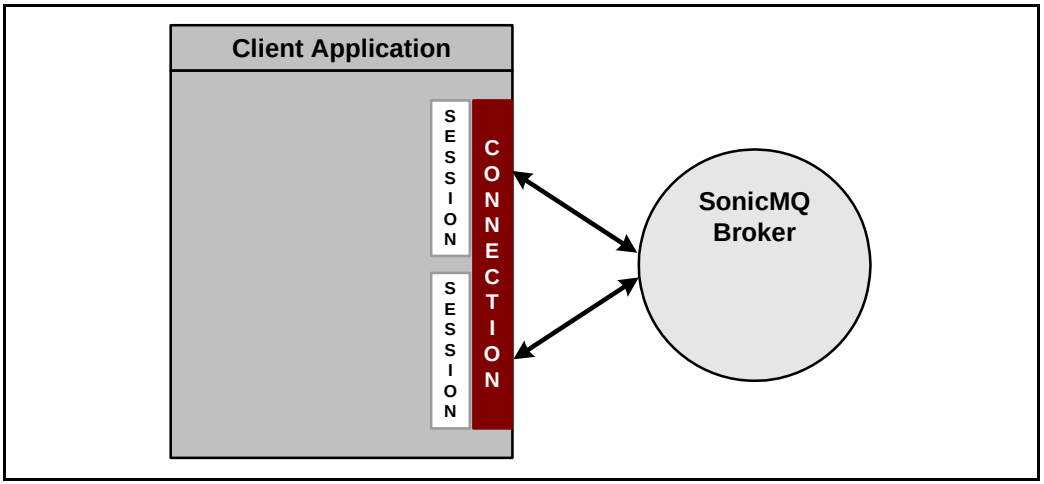


Figure 13. Multiple Sessions on a Connection

Creating Session Objects and the Listeners

Two sections—"Creating and Monitoring a Connection" on page 90 and "Handling Exceptions on the Connection" on page 91—provide information and examples of setting up connections. Once you have a connection, you can create session objects and listeners.

Code Sample 3 shows a continuation of the `ReliableChat` sample from the section "Creating and Monitoring a Connection" on page 90. Two sessions are created—one session to work with the standard input and send functions, and the other to work with the message listener and the messages it delivers for consumption. Each session declares its acknowledgement mode, then sets up the destination and the publisher or subscriber. The message listener is activated against the consumer destination.

Code Sample 3. ReliableChat: Create Session Objects and Listeners

```
try {
     try {
          // Register this class as the exception listener for any problems.
          connect.setExceptionListener((Sonic.Jms.ExceptionListener) this);
          }
          catch (Sonic.Jms.JMSException jmse) {
               System.Console.Error.WriteLine("Cannot set onException listener.");
               System.Console.WriteLine(jmse.StackTrace);
               System.Environment.Exit(1);
        }

        pubSession = connect.createSession(false,Sonic.Jms.Session.CLIENT_ACKNOWLEDGE);
        subSession = connect.createSession(false,Sonic.Jms.Session.CLIENT_ACKNOWLEDGE);
        Sonic.Jms.Topic topic = pubSession.createTopic (APP_TOPIC);
        Sonic.Jms.MessageConsumer subscriber = subSession.createDurableSubscriber(topic,
                                                                "SampleSubscription");
        subscriber.setMessageListener(this);
        publisher = pubSession.createProducer(topic);

....
```

Starting the Connection

When all the session objects and settings are established, the `ReliableChat` connection is started:

```
connect.start( );
```

Messages are composed and sent by the publisher session. Messages are delivered and consumed by the subscriber session. See "Connections" on page 89 for information about setting up and working with connections.

Messaging Domains

Messaging domains are differentiated by messaging behaviors. The programming functionality for the domains is similar, as shown in the interfaces and methods in Table 5.

Table 5. Connected Session Functionality Common to PTP and Pub/Sub

Sonic.Jms and/or Sonic.Jms.Ext Interfaces	***Functionality in Either Domain***
ConnectionFactory	• Allows administrative control of communication resources. • Creates one or more `Connections`.
Connection	• Creates one or more `Sessions`. • Supports concurrent use. • Lets applications specify name-password for client authentication. • Allows unique client identifiers. • Provides `ConnectionMetaData`. • Supports an `ExceptionListener`. • Provides `start( )` and `stop( )` methods. • Provides a `close( )` method for connections.
Session	• Serves as a factory for `MessageProducers` and `MessageConsumers`. • `Sessions` and `Destinations` are used to create multiple `MessageProducers` and `MessageConsumers`. • Serves as a factory for `TemporaryDestinations`. • Creates `Destination` objects with dynamic names. • Serves as a factory for `Messages`. • Supports serial order of messages consumed and produced. • Retains consumed messages until acknowledged. • Serializes execution of registered `MessageListeners`. • Provides a **`close( )`** method for sessions.

Chapter 6 Messages

This chapter provides information about creating and handling messages in SonicMQ, and contains the following sections:

About Messages

A SonicMQ message encapsulates a message body as its payload and exposes metadata that identifies, at a minimum, the message and its timestamp, destination, and priority.

When a text message is published, it might be coded as:

```
private void jmsPublish (String aMessage)
    Sonic.Jms.TextMessage msg = session.createTextMessage();
    msg.setText( user + ": " + aMessage );
    publisher.publish( msg );
```

When a message is received it might be through an asynchronous listener:

```
// Handle an asynchronously received message
public void onMessage( Sonic.Jms.Message aMessage)
{
    ...
    // Cast the message as a text message.
    Sonic.Jms.TextMessage textMessage = (Sonic.Jms.TextMessage) aMessage;
    // Read a single String from the text message, print to stdout.
    String string = textMessage.getText();
    ...
}
```

Message Type

All message types extend the **`Message`** interface, which also defines message headers and the **`acknowledge( )`** method used by all messages. Figure 14 lists the SonicMQ message types.

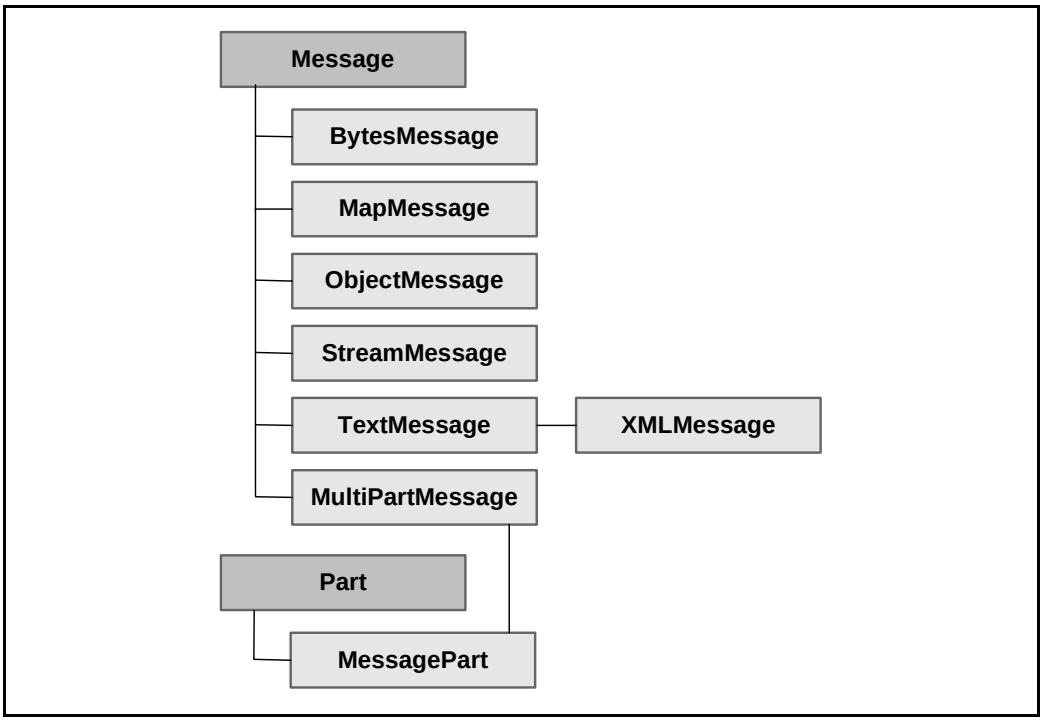


Figure 14. SonicMQ Message Types

The message types are defined as follows:

- **Message** — The root interface of all messages can be used for a bodyless message. All the standard message metadata is available—the header fields and the properties.
- **BytesMessage** — The body is a stream of uninterpreted bytes. This message type exists to support cases where the contents of the message is shared with applications that cannot read C# types or 16-bit Unicode encodings. It is also useful when the information to send already exists in binary form.

- **MapMessage**— The body is a set of name-value pairs where names are strings and values are C# primitive types. The entries can be accessed sequentially or randomly by name. An example of `MapMessage` usage is a message describing a new product, which includes the price, weight, and description; the names in the `MapMessage` correlate to columns in a database table in which the consumer stores the information.
- **ObjectMessage** — The body contains a serializable C# object. An `ObjectMessage` is useful when both clients are C# applications with access to the same class definition.
- **StreamMessage** — The body is a stream of C# unkeyed primitive values that is filled and accessed sequentially. Since a `StreamMessage` contains only raw data and no keys, it takes up less space than an equivalent `MapMessage`.
- **TextMessage** — The body is a `String`. Use a `TextMessage` when the message content does not require any particular structure, for example, when the message body is simply printed or copied by the consumer.
- **XMLMessage** — The body contains valid XML.
- **MultipartMessage** — The body is composed of one or more parts. There are methods to add, delete, and get the constituent parts. The parts might be `MessageParts`, `Sonic.Jms.Message` implementations in addition to primitive types such as XML, HTML, or any type in MIME format such as `text/xml`.

Creating a Message

Create a message *type* from a session method in the form:

```
Sonic.Jms.[type]Message msg = session.create[type]Message()
```

Use the following session methods to create different types of messages:

- `Sonic.Jms.Message msg = session.createMessage( )`
- `Sonic.Jms.BytesMessage msg = session.createBytesMessage( )`
- `Sonic.Jms.MapMessage msg = session.createMapMessage( )`
- `Sonic.Jms.ObjectMessage msg = session.createObjectMessage( )`
- `Sonic.Jms.StreamMessage msg = session.createStreamMessage( )`
- `Sonic.Jms.TextMessage msg = session.createTextMessage( )`

The **`MultipartMessage`** interface extends **`Message`**; the **`XMLMessage`**, interface described in the following section, extends **`TextMessage`**.

Working With Messages That Have Multiple Parts

Many applications—especially those dealing with document-centric business messaging like SOAP 1.1 with Attachments—focus on documents that have multiple parts, each of which might be differentiated by standard MIME content typing. SonicMQ lets you treat messages as parts of a multipart message and include one multipart message inside another.

The following figure shows the structure of a `MultipartMessage`:

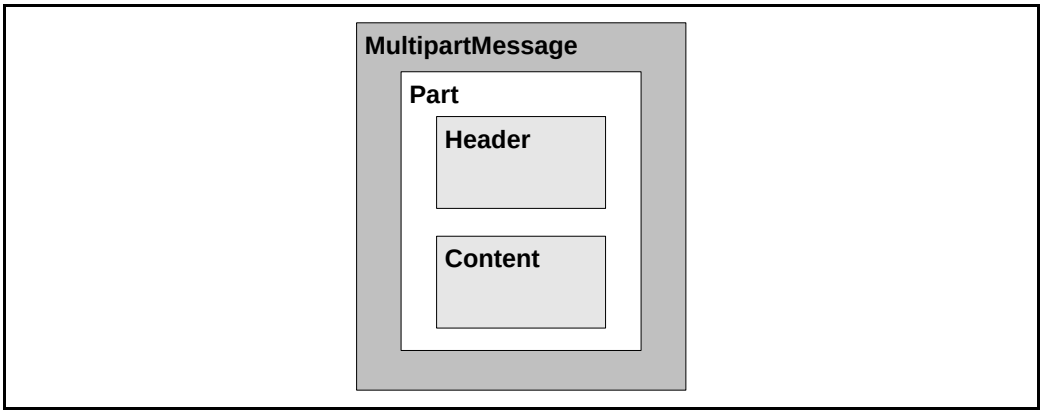


Figure 15. Structure of a MultipartMessage

Each `MultipartMessage` can have properties and zero or more parts. Each part has content and a header that declares at least the part's content type.

`MultipartMessages`, their headers, and their parts are interfaces:

- `Sonic.JMS.Ext.`**`MultipartMessage`**
- `Sonic.JMS.Ext.`**`Part`**
- `Sonic.JMS.Ext.`**`Header`**

MultipartMessage Type

The `MultipartMessage`, a subclass `Sonic.Jms.Message`, is limited to 10 megabytes. It must be completely created on the producer, and must be sent to the broker as a single logical transfer.

Producing a MultipartMessage

To produce a `MultipartMessage`, create the parts and add them to an instance of a `MultipartMessage`. The following example describes sending objects as message parts.

When wrapping a message in a `MultipartMessage` the entire message is wrapped, including the message header and properties. This process provides a technique for handling undeliverable or indoubt messages. If a message needs to be re-routed, it can be packaged in a `MultipartMessage` with the problem message as a `Part` and routed to a special destination for analysis and processing. The following code excerpts are from the `MultipartMessage` sample application, describing the assembly of a multipart message:

- `part1` is a `TextMessage`:

```
Sonic.Jms.TextMessage msg1 = session.createTextMessage();
msg1.setText(" this is a TextMessage "  );
Sonic.Jms.Ext.Part part1 = mm.createMessagePart(msg1);
part1.getHeader().setContentId("CONTENTID1");
```

- The part is added to the `MultipartMessage` and then sent:

```
mm.addPart(part1);
sender.send(mm);
```

The producer methods in the `MultipartMessage` interface are listed in Table 6.

Table 6. Producer Methods in the MultipartMessage Interface

Method	*Description*
`Part` **`createPart`**`()`	Creates an empty part.
`Part` **`createPart`**`(Object` *`object`*`, String` *`type`*`)`	Creates a part where *`type`* is the ContentType associated with the content.
`Part` **`createMessagePart`**`(Message` *`message`*`)`	Creates a part whose content is a `Message` where *`message`* is the content of the part.
`void` **`addPart`**`(Part` *`part`*`)`	Adds *`part`* to the `MultipartMessage` at the end of the message.
void **`addPartAt`**`(Part` *`part`*`, int` *`index`*`)`	Adds *`part`* to the `MultipartMessage` at position *`index`*.

Table 6. Producer Methods in the MultipartMessage Interface (*continued*)

Method	*Description*
void **`removePart`**(String *`cid`*)	Removes part with content-ID *`cid`* from the `MultipartMessage`.
void **`removePart`**(int *`index`*)	Removes part *`index`* from the `MultipartMessage`.

Consuming a Multipart Message

A `MultipartMessage` is held by brokers and routed over routing nodes with the same integrity as any other `Sonic.Jms.Message`. A client receiving the `MultipartMessage` can recover the original message from the `Part`. An original message that is copied into a part has its own, original header fields and properties.

Note When a message is in read-only mode—after it has arrived at a `MessageListener`—the set methods on the `Part` and `Header` return errors.

Receiving a `MultipartMessage` is the same as any other message. Consuming the message is done with standard `MessageListeners` or calls to `receive( )`.

The following code excerpt is from the `MultipartMessage.cs` sample where the `onMessage` method is delivered a `MultipartMessage`, then passes it through its `unpackMM` pattern to determine how many parts the message contains. The sample application then iterates through the handling of each part:

```
private void unpackMM(Sonic.Jms.Message aMessage, int depth)
{
    int n = depth;
    ...
    try
    {
        indent(n); System.Console.Out.WriteLine ("Extend_type property = " +
                                aMessage.getStringProperty(Constants.EXTENDED_TYPE));
        MultipartMessage mm = (MultipartMessage)aMessage;
        int partCount = mm.getPartCount();

        indent(n);
        System.Console.Out.WriteLine ("partCount of this MultipartMessage = " + partCount);
        for (int i = 0; i < partCount; i++)
        {
            System.Console.Out.WriteLine();
            indent(n); System.Console.Out.WriteLine("Begin part " + (i+1));
            Part part = mm.getPart(i);
```

Each part is evaluated to see if it should be treated as a message part or evaluated as a MIME content type:

```
            if (mm.isMessagePart(i))
            {
                Sonic.Jms.Message msg = mm.getMessageFromPart(i);
                if (msg is MultipartMessage)
                    unpackMM(msg, ++depth);
                else
                    unpackJMSMessage(msg, n);
                }
                else
                {
                    unpackPart(part, n);
                }
                indent(n); System.Console.Out.WriteLine("end of part " + (i+1));
                System.Console.Out.WriteLine();
            }
```

The methods for the `MultipartMessage` consumer are listed in Table 7.

Table 7. Consumer Methods in MultipartMessage Interface

Method	*Description*
`void` **`clearReadOnly`**`()`	Makes the message writable.
`String` **`getProfileName`**`()`	Returns the extended type or profile used to create this message.
`boolean` **`doesPartExist`**`(String` *`cid`*`)`	Tests whether a part with the content-id *`cid`* exists.
`boolean` **`isMessagePart`**`(int` *`index`*`)`	Tests whether part with *`index`* is a `MessagePart`
`boolean` **`isMessagePart`**`(String` *`cid`*`)`	Tests whether the part with content ID *`cid`* is a `MessagePart`.
`int` **`getPartCount`**`()`	Returns the number of parts in the `MultipartMessage`.
`Part` **`getPart`**`(int` *`index`*`)`	Gets part *`index`* of the message.
`Part` **`getPart`**`(String` *`cid`*`)`	Gets the part of the message identified as content ID *`cid`*.
`Message` **`getMessageFromPart`**`(int` *`index`*`)`	Gets a message from part *`index`* of the message.
`Message` **`getMessageFromPart`**`(String` *`cid`*`)`	Gets a message from the part of the message with the content ID *`cid`*.
`boolean` **`isReadOnly`**`()`	Tests whether a message is read only.

JMS_SonicMQ_ExtendedType Property

A `MultipartMessage` is not only identified as an instance of `MultipartMessage`. A string property, `JMS_SonicMQ_ExtendedType`, is also set when a `MultipartMessage` type is sent to carry the profile of the message. The name is also accessible in `Sonic.Jms.Ext.Constants` as:

```
public String EXTENDED_TYPE = “JMS_SonicMQ_ExtendedType”
```

Parts of a MultipartMessage

Each part of a `MultipartMessage` has an associated `Header` and content.

- The `Header` contains name/value pair to represent header objects such as `ContentType` and `ContentId`. The `Header` can be implemented separately from a `MessagePart`, or as methods on the `Part` itself.
- The content of a `Part` is accessed as one of the following:
  - An input stream by using the `getInputStream( )` method.
  - C# object by using the `getContent( )` method.

The methods in the `Parts` interface are listed in Table 8.

Table 8. Methods in the Parts Interface

Method	*Description*
`setContent``(Object` *`object`*`, string` *`type`*`)`	Sets the part's content as a C# object of content type *`type`*.
`setContent``(byte[]` *`content`*`)`	Sets the part's content as a byte array of *`content`*.
`Object` **`getContent`**`()`	Returns the content of the part as a C# object.
`byte[]` **`getContentBytes`**`()`	Returns the content of the part as a byte array.
`Header` **`getHeader`**`()`	Returns the `Header` for the `Part`.
`System.IO.InputStream` **`getInputStream`**`()`	Returns an input stream for this part's content
`System.IO.OutputStream` **`getOutputStream`**`()`	Returns an output stream for writing this part's content.

MessagePart Subclass

A subclass of the `Part` is the `MessagePart`, used by an application to wrap one or more messages into a `MultipartMessage`. The `ContentType` of SonicMQ `MessageParts` is set implicitly, as shown in Table 9.

Table 9. Implicit Content-Type for SonicMQ Message Types

Message Type	*Content-Type*
`Message`	`application/x-sonicmq-message`
`BytesMessage`	`application/x-sonicmq-bytesmessage`
`MapMessage`	`application/x-sonicmq-mapmessage`
`ObjectMessage`	`application/x-sonicmq-objectmessage`
`StreamMessage`	`application/x-sonicmq-streammessage`
`TextMessage`	`application/x-sonicmq-textmessage`
`XMLMessage`	`application/x-sonicmq-xmlmessage`
`MultipartMessage`	`application/x-sonicmq-multipartmessage`

The `MessagePart` interface inherits all its methods from the `Part` interface.

Header of the MultipartMessage or a Part

The `MultipartMessage` itself and each `Part` have `Header` objects associated with them that hold the `ContentId`, `ContentType`, and other fields (typically MIME). Table 10 lists the methods in the `Header` interface.

Table 10. Methods in the Header Interface

Method	*Description*
`void` **`setContentId`**`(string` *`cid`*`)`	Sets the `ContentId` header of the message or attachment part to the value *`cid`*.
`setContentType``(string` *`type`*`)`	Sets the `ContentType` header of the message or attachment part to the value *`type`*.

Table 10. Methods in the Header Interface (*continued*)

Method	*Description*
`void` **`setHeaderField`**`(string` *`name`*`, string` *`value`*`)`	Sets the value of a header field *`name`* to the string *`value`*.
`string` **`getContentId`**`()`	Returns the `ContentId` of the message or attachment part.
`string` **`getContentType`**`()`	Returns the `ContentType` of the message or attachment part. Returns the `ContentType` of the part or null.
`System.Collections.IEnumerator` **`getHeaderFieldNames`**`()`	Returns the list of all header fields.
`string` **`getHeaderField`**`(string` *`name`*`)`	Returns the value of header field *`name`* or null if it does not exist.
`string` **`getHeaderField`**`(string` *`name`*`, string` *`value`*`)`	Returns the value of a header field name. If it does not exist, defaults to *`value`*.
`void` **`removeHeaderField`**`(string` *`name`*`)`	Removes header *`name`* from the part.
`void` **`removeAllHeaders`**`()`	Removes all headers.

Message Structure

Messages are composed of the following parts:

- **Header Fields** — All messages support the same set of header fields. Header fields contain values used by clients and brokers to identify and route messages.
- **User-defined Properties** — User-defined name-value pairs that can be used for filtering and application requirements.
- **Provider-defined Properties** — Properties defined and typed by SonicMQ for carrying information used by SonicMQ features.
- **Supported JMS-defined Properties (JMSX)** — Predefined name-value pairs that are an efficient mechanism for supporting message filtering.
- **Body** — There are several types of message bodies, which cover the majority of messaging styles currently in use.

Note While the SonicMQ message system provides programmatic access to all components of a message, message selectors and routing data are constrained to the header fields and properties, not the message body.

Message Header Fields

The message header fields are defined and used by the sender and the broker to convey basic routing and delivery information. The message header fields are described in detail in Table 11.

Table 11. Message Header Fields

Header Field	*Type*	*Description*	*Usage*	*Comments*
JMSDestination Required. Set by the producer send/publish method.	`String`	The destination where the message is sent.	While a message is being sent, this value is ignored. After completion of the `publish|send` method, it holds the destination specified by the send.	When a message is received, its destination value must be equivalent to the value assigned when it was sent.
JMSDeliveryMode Required. Set in a producer send/publish parameter.	`String`	Specifies whether the message is to be retained in the broker's persistent storage mechanism.	Required. Must be `PERSISTENT`, `NON_PERSISTENT`, `NON_PERSISTENT_SYNC`, `NON_PERSISTENT_ASYNC`, or `DISCARDABLE`	Default value is `NON_PERSISTENT`.
JMSMessageID Required. Set by the producer's send/publish method.	`String`	SonicMQ field for a unique identifier.	A message ID value must start with "`ID:`".	While required, the algorithm that calculates the ID on the client can be bypassed, which sets the `JMSMessageID` to `null`.
JMSTimestamp Required. Set by the producer's send/publish method.	`long`	GMT time on the producer's system clock when the message is sent.		Set method exists but is always overridden by the send method valuation.

Table 11. Message Header Fields (*continued*)

Header Field	*Type*	*Description*	*Usage*	*Comments*
JMSCorrelationID Optional. Set by producer method.	`String`	Broker-specified message ID or an application-specific String.	Required to support the native concept of a correlation ID.	An application made up of several clients might want an application-specific value for linking messages.
JMSCorrelationID AsBytes Optional. Set by producer method.	`bytes`	A native `byte[]` value.		
JMSReplyto Optional. Set by producer method.	`String`	The destination where a reply to the current message should be sent.	If `null`, no reply is expected. If not `null`, expects a response, but the actual response is optional and the mechanism must be coded by the developer.	Message replies often use the `CorrelationID` to assure that replies synchronize with the requests.
JMSRedelivered Set by broker.	`boolean`	If `true` it is likely that this message was delivered to the client earlier but the client did not acknowledge its receipt at that time.	Set by the broker at the time the message is delivered. Note that, while `setJMSRedelivered (boolean)` exists, this header field has no meaning on send and is left unassigned by the sending method.	When acknowledgement is expected and not received in a specified time, the broker can decide to set this and resend.
JMSType Optional. Set by producer method.	`String`	Contains the name of a message's definition as found in an external message type repository.	Recommended for systems where the repository needs the message type sent to the application.	This is not, by default, the message type.

Table 11. Message Header Fields (*continued*)

Header Field	*Type*	*Description*	*Usage*	*Comments*
JMSExpiration Required. Set by the producer send/publish method by incrementing the current GMT time on the producer system by the producer send/publish parameter, `timeToLive`.	`long`	When a message's expiration time is reached, the broker can discard it. Clients should not receive messages that have expired, but this is not guaranteed.	The sum of the time-to-live value specified by the client and the GMT at the time of the send. If the time-to-live is specified as `0`, the message does not expire. Default value is `0`.	When a message is sent, expiration is left unassigned. After completion of the `send` method, it holds the expiration time of the message. Default value is `0`. The expiration of a message can be managed by setting the message property `JMS_SonicMQ_preserveUndelivered` which will transfer an expired (or undeliverable) message to the broker's DMQ.
JMSPriority Required. Set in a producer send/publish parameter.	`int`	Sets a value that allows a message to move ahead of other undelivered messages in a topic or queue. Also allows message selectors to pick messages at a given priority.	A ten-level priority value with `0` as the lowest priority and `9` as the highest. `0` to `4` are normal. `5` to `9` are expedited. Default value is `4`.	SonicMQ does not require strict priority ordering of messages. However, the broker does its best to deliver expedited messages ahead of normal messages.

Setting Header Values When Sending/Publishing

The basic method for producing a message allows essential delivery information to accept default values, for example:

```
publisher.publish(Message message)
```

Three of the message header fields have default values as static const variables:

- DEFAULT_DELIVERY_MODE = NON_PERSISTENT
- DEFAULT_PRIORITY = 4
- DEFAULT_TIME_TO_LIVE = 0

The delivery mode default value of NON_PERSISTENT is interpreted as NON_PERSISTENT_SYNC when security is enabled and NON_PERSISTENT_ASYNC when security is not enabled.

The default header field values can be changed in the signature of the send or publish method to override the defaults:

- **Point-to-point:**

```
sender.send( Message message,
             int deliveryMode,
             int priority,
             long timeToLive)
```

- **Publish and Subscribe:**

```
publisher.publish(Message message,
             int deliveryMode,
             int priority,
             long timeToLive)
```

If you use this format of the method but do not intend to override some of the default values, you can substitute the values back into the parameter list. For example:

```
private static const int MESSAGE_LIFESPAN = 1800000;
// milliseconds (30 minutes)
sender.send( msg,
             Sonic.Jms.DeliveryMode.PERSISTENT,
             Sonic.Jms.DefaultMessageProperties.DEFAULT_PRIORITY,
             MESSAGE_LIFESPAN);
```

Message Properties

Properties are optional fields that are associated with a message. No message properties are required for any message producer. The property values are used for message selection criteria and data required by applications and other messaging infrastructures. The order of property values is not defined.

Data is handled in a message's body more efficiently than data in a message's properties. For best performance, applications should only use message properties when they need to customize a message's header. The primary reason for doing this is to support customized message selection.

Property names must obey the rules for a message-selector identifier. Property values can be `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, and `string`.

Property values are set prior to sending a message. When a client receives a message, its properties are in read-only mode. If **`clearProperties`** is called, the properties are erased and then can be set.

Provider-defined Properties (JMS_SonicMQ)

SonicMQ reserves some property names and declares each property's type. The following properties are prescribed in SonicMQ for use in expressing intended handling of undelivered messages, setting preferred message encryption, and indicating message types.

Table 12 lists SonicMQ-defined properties.

Table 12. SonicMQ Provider-defined Properties

Function	*Provider-defined Property*	*Type*	*Set by*
Handling of undeliverable messages	`JMS_SonicMQ_preserveUndelivered`	`boolean`	Producer
	`JMS_SonicMQ_notifyUndelivered`	`boolean`	Producer
	`JMS_SonicMQ_undeliveredReasonCode`	`int`	Broker
	`JMS_SonicMQ_undeliveredTimestamp`	`long`	Broker
QoP setting	`JMS_SonicMQ_perMessageEncryption`	`boolean`	Producer
Message type	`JMS_SonicMQ_Extended_Type`	`String`	Producer

See Chapter 10, "Guaranteeing Messages," for detailed information about how these properties contribute to handling undeliverable messages in local brokers and dynamic routing nodes.

Per Message Encryption

SonicMQ brokers can establish Quality of Protection (QoP) settings on a security-enabled broker so that a client application producing to a destination must try to send the message after the encryption and integrity requested has been performed. The client application is not aware of the QoP enforced on it by the destination. Similarly, per-message encryption does not force the broker to encrypt a message to a message consumer when the destination's QoP settings do not require it.

When an application wants to be sure that it sends messages to a security-enabled broker after encrypting them and establishing integrity tests, the application can set the property, `JMS_SonicMQ_perMessageEncryption`. On a broker that is not security-enabled, this setting is a no-op.

The property that selects per message encryption is a boolean property:

```
JMS_SonicMQ_perMessageEncryption=true
```

This setting can also be set by using a constant:

```
aMessage.setBooleanProperty
              (Sonic.Jms.Ext.Constants.ENCRYPT_MESSAGE, true);
```

You can determine whether a broker is security enabled by calling the `Sonic.Jms.Ext.Connection.isSecure( )` method:

```
if (connect.isSecure())
{
    aMessage.setBooleanProperty
              (Sonic.Jms.Ext.Constants.ENCRYPT_MESSAGE, true);
}
else
{
    //Handle condition where broker is insecure...
}
```

JMS-defined Properties (JMSX)

The **JMSX** property name prefix is reserved for optional JMS-defined properties. Properties that are set on send are available to the producer and the consumers of the message.

Properties can be referenced in message selectors whether or not they are supported by a connection. They are treated like any other absent property. Table 13 lists and describes the JMSX message properties used in SonicMQ. These JMSX properties are set by the producer.

Table 13. JMSX Properties Used in SonicMQ

JMSX Property	*Type*	*Set by*
JMSXGroupID	String	Producer on send
JMSXGroupSeq	int	Producer on send
JMSXUserID	String	Broker

For more information about using JMSXUserID in basic authentication, see the *Progress SonicMQ Deployment Guide*.

User-defined Properties

A message supports application-defined property values, providing a mechanism for adding application-specific header fields to a message. For example:

- Identifiers for audits or reconcilement of undeliverable messages. These might be properties you define, such as OriginatorHostID, AuditID, RealTimeDeviceID, RFID, or similar.
- Hints for rerouting undeliverable messages such as ReturnURL, AlternateURL, ReturnDestination, ReturnEmail, or similar.
- Settings used by SonicMQ's outbound routing to provide the security and attributes for routing pure HTTP messages to HTTP Web servers. These are described in the "Using HTTP(S) Direct" part of the *Progress SonicMQ Deployment Guide*. Examples of such properties are:
  - X-HTTP-DestinationURL to specify the target URL.
  - X-HTTP-AuthUser and X-HTTP-AuthPassword for Web server authentication
  - X-HTTP-ReplyAsSOAP, X-HTTP-RequestTimeout, X-HTTP-Retries, and X-HTTP-RetryInterval are not attached to the HTTP message as header properties. They

define the HTTP Direct outbound routing connection attempts and, in the case of `X-HTTP-ReplyAsSOAP`, the reply format of internally generated error replies.

- `X-HTTP-GroupID` to define message grouping for ordered delivery
- SSL-related properties for HTTPS Web server authentication:

  ```
  X-HTTPS-CipherSuites, X-HTTPS-CACertificatePath,
  X-HTTPS-ClientAuthCertificate, X-HTTPS-PrivateKey,
  X-HTTPS-PrivateKeyPassword, X-HTTPS-ClientAuthCertificateForm,
  ```

Determining the Pending Queue for Messages

SonicMQ brokers maintain thread pools for outbound HTTP Direct messages so that messages can be grouped by URL. Each thread uses a reserved pending queue. Two techniques enable multiple pending queues to operate concurrently:

- When a client application sends messages to a node with the property **`X-HTTP-GroupID`** set to a string so that many applications using that GroupID have their messages dispatched in the order they were submitted by the applications.
- When a routing definition has the option **Group Messages by URL** selected and GroupIDs are not in use, messages routing through an HTTP Direct routing node use the value of **`X-HTTP-DestinationURL`** on the message to group messages for the same destination, sending them through the same pending queue after normalizing the URL into patterns.

For more information, see the “Grouping Messages by Destination URL” section of the “HTTP(S) Direct Acceptors and Routings” chapter in the *Progress SonicMQ Deployment Guide*.

The active pending queues can be monitored through the Sonic Management Console’s **Manage** tab where a broker’s routing statistics can be viewed. For more information, see the “Routing Statistics” section of the “Managing SonicMQ Broker Activities” chapter in the *Progress SonicMQ Configuration and Management Guide*.

Setting Message Properties

Message properties are in no specified order. They might or might not contain values or data extracted from the message body. There are no default properties.

Property Methods

JMSX properties can be referenced in message selectors whether or not they are supported by a connection. If values for these properties are not included, they are treated like any other absent property.

User-defined properties are not typed. Data typing is defined by the set method used, such as `setIntProperty( )`.

When data is retrieved, the `get( )` method for user-defined properties can attempt to coerce the data into that data type when the value is retrieved.

The setting and getting of message properties allows a full range of data types when the property is established. The properties can be retrieved as a list. A property value can be retrieved by using a `get( )` method for the property name.

Checking Whether a Property Exists

Use the `propertyExists( )` method to check whether a property value exists:

```
public boolean propertyExists(String name)
```

where `name` is the name of the property to test. Returns **`TRUE`** if it exists.

Clearing Message Properties

Use the **`clearProperties`** method to delete a message's properties. This method leaves the message with an empty set of properties. Clearing properties affects only those properties that are defined and has no impact on the header fields or the message body:

```
public void clearProperties()
```

Setting the Property Type

Message properties are set as name-value pairs where the value is of the declared data type. Setting a property type that does not exist causes that property type to exist as a property in that message:

```
set[type]Property(String name, [type] value)
```

where *type* is one of the following:

```
{ Boolean | Byte | Short | Int | Long | Float | Double | String }
```

For example:

```
setBooleanProperty(“reconciled”,true).
```

Getting Property Names

Use **`getPropertyNames`**`()` to retrieve a property name enumeration. Use this enumeration to iterate through a message's property values. Then use the various property `get( )` methods to retrieve their respective values.

Getting Property Values

Use the `get[type]Property( )` method to get the value of a property. If the property does not exist, a `null` is returned:

```
public [type] get[type]Property(String name);
```

where *type* is one of the following:

```
{ Boolean | Byte | Short | Int | Long | Float | Double | String }
```

For example, `boolean getBooleanProperty(“reconciled”)` returns `true`.

Property values can be coerced. The accepted conversions are listed in Table 14 where a value written as the row type can be read as the column type. For example, a `short` property can be read as a short or coerced into an `int`, `long` or `String`. An attempt to coerce a `short` into another data type is an error.

Table 14. Permitted Type Conversions for Message Properties

	boolean	*byte*	*short*	*int*	*long*	*float*	*double*	*String*
boolean	Yes	No	No	No	No	No	No	Yes
byte	No	Yes	Yes	Yes	Yes	No	No	Yes
short	No	No	Yes	Yes	Yes	No	No	Yes
int	No	No	No	Yes	Yes	No	No	Yes
long	No	No	No	No	Yes	No	No	Yes
float	No	No	No	No	No	Yes	Yes	Yes
double	No	No	No	No	No	No	Yes	Yes
String	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Valid coercions are indicated with `Yes`. Intersections marked with `No` throw a `JMSException`. A string-to-primitive conversion might throw a run-time exception if the primitives **`valueOf`**`()` method does not accept it as a valid string representation of the primitive.

Message Body

The message body has no default value and is not required to have any content. The message body is populated by the `set( )` method for the message type. The following sections explain how to use the `set( )` and `get( )` methods for the message body.

Setting the Message Body

Use the `set( )` methods for all message types except `XMLMessage` unless the message is read-only (in which case you must copy or reset the received message). For example, for a `TextMessage`:

```
msg.setText(aMessage);
```

Important If you use `setText(String string)` where *string* is the string containing the message's data, you set the string containing this message's data, overriding `setText` in class `TextMessage`.

For information about setting the `Parts` in a `MultipartMessage`, see "Parts of a MultipartMessage" on page 148.

Getting the Message Body

Use the `get( )` methods for all message types except `XMLMessage`. For example:

```
msg.getText(aMessage);
```

For information about getting `Parts` of a `MultipartMessage` and distinguishing message types from other MIME types, see "MultipartMessage Type" on page 143 and "Decomposing Multipart Messages" on page 48.

Chapter 7 Message Producers and Consumers

This chapter describes the generic programming model for messaging that is common to both messaging models, Publish and Subscribe (Pub/Sub) and Point-to-point (PTP). These two messaging models are described in Chapter 8, "Point-to-point Messaging," and Chapter 9, "Publish and Subscribe Messaging." This chapter covers the following topics:

About Message Producers and Message Consumers

To establish message producers and message consumers in one of the messaging models, you create an appropriate `ConnectionFactory`, then create connections. You create sessions on each connection and then create the session objects, as shown in Figure 16.

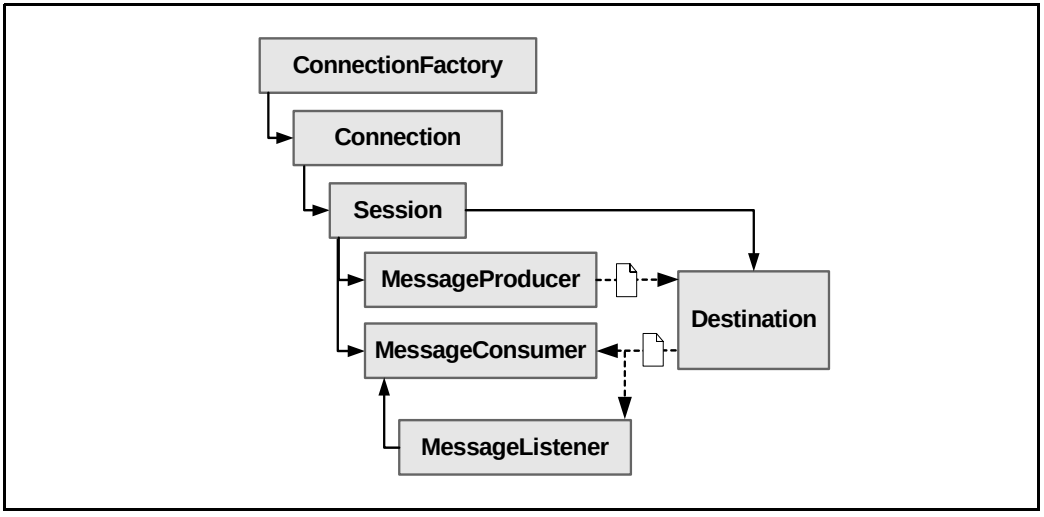


Figure 16. Generic Messaging Model

The message producers send messages to a destination on a broker. Message consumers get messages from a destination by implementing asynchronous `MessageListeners` or by doing synchronous receives.

Message Ordering and Reliability

Various factors in a loosely coupled messaging structure can impact the sequence of messages delivered to consumers. Message ordering and redelivery both contribute to reliable message delivery. Messaging services are impacted by many uncontrollable environmental factors ranging from latency and machine outages to internal factors such as related applications that do not accept data types, values, poorly formed XML data, and data payloads. Message delivery is distinctly nonlinear.

Message ordering and reliability common to all messaging domains are described in this chapter. See also "Message Ordering and Reliability in PTP" on page 197 and "Message Ordering and Reliability in Pub/Sub" on page 220 for details about message ordering and reliability within those domains.

Messages can be delivered with a range of options to modify message ordering and invoke features that improve reliability:

- The producer can set the **time-to-live** of the message so that obsolete messages can expire. If message A is set at one minute, message B at five seconds, and message C at one hour, then after three minutes with no deliveries, only message C still exists. Ordering is maintained while expiration is a user-defined value.
- The producer can set the **delivery mode** of messages so the broker confirms persistent storage of the message before acknowledgement is sent. In the event of a broker failure, a message that the broker acknowledged before it was persisted might be lost. The delivery mode of a message characterizes the message for its entire life. If a non-persistent message is waiting in a durable subscription or a queue when the broker restarts, the message does not exist when the broker comes back up.
- The producer can set the **priority** of a message so that the broker can position a more recent message before an older one.
- The producer uses a synchronous process to put the message on the broker's message store. When released, the message is **acknowledged** as delivered to its interim destination.
- The consumer can use **listeners** to get messages as they are made available.
- Messages sent in the **`NON_PERSISTENT`** delivery mode can arrive prior to messages that are **`PERSISTENT`**.
- The consumer starts a session by expressing its preferred **acknowledgement** mode—transactional or not, explicit or implicit.
- Connections can be monitored and, when broken, automatically attempt to **reconnect**. (This might not be necessary if you are using fault-tolerant connections. See "Fault-Tolerant Connections" on page 93.)
- Message senders in the Internet environment are not guaranteed consistent communication times. Transmission **latencies** can cause messages to be produced before other messages. As a result, two messages from two sessions are not required—and cannot be reliably expected—to be in any specific sequence.

Destinations

Destinations are objects that provide the producer, broker, and consumer with a context for delivery of messages. Destinations can be dynamic objects created as needed (topics only) or temporary objects created for very limited use.

For topics, SonicMQ provides extended management and security with **hierarchical name spaces**; for example, **`jms.samples.chat`**. See Chapter 11, "Hierarchical Name Spaces," for more information.

Important Table 2, "Restricted Characters for Names" on page 83 lists characters that are not allowed in SonicMQ names. Refer to this list for restricted characters must not use in your topic or queue names.

The following restrictions apply to queue and topic names:

- The strings `$SYS` and `$ISYS` are reserved for administrative queues. See the *Progress SonicMQ Configuration and Management Guide* for more information.
- The destination name length limit is 256 characters.
- A queue name cannot begin with the string "SonicMQ." This prefix is reserved for system queues. Queues whose names begin with "SonicMQ" cannot be added or deleted.

You can programmatically store and retrieve defined destinations.

Steps in Message Production

Every time a `Session` wants to send a message to a `Destination`, it must create a `MessageProducer`. The following sections explain the steps required to produce a message within a connected `Session`, using the approach implemented in the Chat sample:

1. “Create a Session” on page 169.
2. “Create the Producer on the Session” on page 170.
3. “Create the Message Type and Set Its Body” on page 171.
4. “Set Message Header Fields” on page 172.
5. “Set the Message Properties” on page 172.
6. “Elect Per Message Encryption” on page 173 (optional).
7. “Produce the Message” on page 173.

Create a Session

After establishing a connection, the Chat sample creates a `Session`:

```
Sonic.Jms.Session pubSession;
...
pubSession =
connect.createSession(false,Sonic.Jms.SessionMode.AUTO_ACKNOWLEDGE);
```

You can use a `Session` for PTP messaging, Pub/Sub messaging, or both. The Chat sample names the `Session` **`pubSession`**, because it is intended for Pub/Sub messaging.

Create the Producer on the Session

The Chat example sets up the variable `APP_TOPIC` (assigned the value `"jms.samples.chat"`) as the working `Topic` and creates a `MessageProducer` associated with that `Topic`:

```
private const System.String APP_TOPIC = "jms.samples.chat";
...
private Sonic.Jms.MessageProducer publisher = null;
...
Sonic.Jms.Topic topic = pubSession.createTopic (APP_TOPIC);
...
publisher = pubSession.createProducer(topic);
```

`MessageProducer` objects can send messages to any `Destination`, both `Queues` and `Topics`. Here, a `Topic` is the `Destination` passed to the `createProducer( )` method. When a valid `Destination` is passed to the `createProducer( )` method, the returned `MessageProducer` object uses that `Destination` as its default.

The `MessageProducer` object's `send( )` method (the form that specifies no target `Destination`) uses the default `Destination` as its target Destination. You can explicitly specify a different target `Destination` if you use a different form of the `send( )` method.

Create the Message Type and Set Its Body

The Chat example constructs a text message from the standard input (the keyboard) and reads the message in with the **`readLine( )`** method. It creates a new SonicMQ `TextMessage` and sets the text into the message, prepended in the sample by the username, a colon, and a space:

```
try {
     // Read all standard input and send it as a message.
     System.IO.Stream  stdin = System.Console.OpenStandardInput();
     System.IO.StreamReader stdinReader =
                    new System.IO.StreamReader(stdin);
     System.Console.Out.WriteLine("\nEnter text messages to clients that
          subscribe to the " + APP_TOPIC + " topic." + "\nPress Enter to publish
          each message.\n");
     while (true) {
          System.String s = stdinReader.ReadLine();

          if ((System.Object) s == null)
               exit();
          else if (s.Length > 0) {
               Sonic.Jms.TextMessage msg = pubSession.createTextMessage();
               msg.setText(username + ": " + s);
               msg.setJMSDeliveryMode(Sonic.Jms.DeliveryMode.PERSISTENT);
               publisher.send(msg);
          }
     }
}
```

When the sample is run, if the user `Sales` enters `“Hello.”`, the message content is `“Sales: Hello.”`

Set Message Header Fields

The Chat example does not set any message header fields. If you want to change header fields, use the `set( )` methods for message header fields that are available for change:

```
setJMSType("CentralFiles")
```

For some header field `set( )` methods (such as **`setJMSMessageID( )`** and **`setJMSTimestamp( )`**), the value you assign is overwritten at the time the message is produced.

The header fields that are named and typed and also available for assignment are:

- **`JMSCorrelationID`**, reserved for message matching functions
- **`JMSReplyto`**, reserved for request reply information
- **`JMSType`**, available for general use

Set the Message Properties

The Chat example does not set any message properties. If you want to set message properties, use the `set( )` methods for the data type of a property and then supply the property name and its value of the declared type:

```
set[type]Property(String name, String value)
```

For example:

```
setLongProperty("OurInfo_AuditTrail","6789")
```

Elect Per Message Encryption

Destinations can be configured on a broker to encrypt messages. When a producer binds to a destination, the producer is instructed to encrypt or not encrypt by the broker. But this decision for encryption is not revealed to the application. A client application can ensure that a message is sent encrypted to a security-enabled broker by electing to do per message encryption as follows:

```
setBooleanProperty(Sonic.Jms.Ext.Constants.ENCRYPT_MESSAGE, true);
```

Produce the Message

When the message is assigned its attributes (header fields and properties) and its payload, the message is ready to be sent to its destination. The Chat example uses the simplest form of the `send( )` method to send the message to its `Destination`:

```
publisher.send(msg);
```

The form of `send( )` used in the `DurableChat` sample application sets three important message parameters at the moment the `send( )` method is executed:

```
private const long MESSAGE_LIFESPAN = 1800000;
publisher.send(msg,
                  Sonic.Jms.DeliveryMode.PERSISTENT,
                  Sonic.Jms.DefaultMessagePriorities.DEFAULT_PRIORITY,
                  MESSAGE_LIFESPAN);
```

This form of the `send( )` method passes along either the default values or the entered values for:

- **`JMSDeliveryMode`** is [**`NON_PERSISTENT|PERSISTENT|NON_PERSISTENT_SYNC|NON_PERSISTENT_ASYNC|DISCARDABLE`**]
- **`JMSPriority`** is [**`0...9`**] where **`0`** is lowest, **`9`** is highest, **`4`** is the default.
- **`timeToLive`**, the message lifespan that calculates the `JMSExpiration`, is [**`0...n`**] where **`0`** is “forever” and any other positive value **`n`** is in milliseconds.

The `send( )` method assigns—and overwrites, if previously assigned—data to the following header fields:

- **`JMSDestination`**, the producer's current destination
- **`JMSTimestamp`**, based on the producer's system clock
- **`JMSMessageID`**, based on the algorithm run on the producer's system
- **`JMSExpiration`**, based on the producer's system clock plus the `timeToLive`

The release of the synchronous block by the broker returns only a boolean indicating whether the message production completed successfully.

Important While the `JMSExpiration` is calculated from the client system clock at the time of the send, it is enforced on the broker's clock. To accommodate variances between client and broker clocks, the broker adjusts the message expiration to its clock. When the message is forwarded to another broker, the remaining `timeToLive` value (expiration minus current broker GMT time) is forwarded. The time that elapses until the first packet of the message in transit is received is effectively ignored.

Message Management by the Broker

A message at a destination behaves according to the parameters of the message `send` (PTP) or `publish` (Pub/Sub) event. Table 15 lists those parameters and how they direct the broker to handle the message.

Table 15. How Message Producer Parameters Influence the Broker

Producer Parameter	*How the Parameter is Treated by the Broker*
deliveryMode	`deliveryMode = PERSISTENT` — Stores the message in the broker's message log in case of impending failure. Acknowledges the producer only after logging the message. `deliveryMode = NON_PERSISTENT` — If the message is enqueued or stored for a durable subscriber on a broker that shuts down, the message is volatile. This parameter is interpreted as `NON_PERSISTENT_ASYNC` or `NON_PERSISTENT_SYNC` based on whether security is enabled. `deliveryMode = NON_PERSISTENT_ASYNC` — Message publisher methods do not expect any acknowledgement whatsoever. This is the default nonpersistent delivery mode when security is not enabled. Messages can be lost if client fails. `deliveryMode = NON_PERSISTENT_SYNC` — This is the default nonpersistent delivery mode when security is enabled. Message publisher methods block to await acknowledgement. `deliveryMode = DISCARDABLE` — For nontransacted Pub/Sub only. Delivers all messages to subscribers that are keeping up with the flow of messages, but drops the oldest messages waiting for lagging subscribers when new messages arrive, under any of the following conditions: ● When the message server's internal buffers for that subscriber session are full ● When a neighbor cluster member containing a Topic subscription is unavailable and a subscriber is located on the other cluster member ● When an intended durable subscriber is unavailable **NOTE:** A message's deliveryMode is effective throughout its lifespan. If a `NON_PERSISTENT` message is enqueued (PTP) or stored for a durable subscriber (Pub/Sub) on a broker that shuts down, the message is volatile. This behavior stays with a message throughout its travels in a dynamic queue routing deployment, and even applies in the dead message queue.

Table 15. How Message Producer Parameters Influence the Broker (*continued*)

Producer Parameter	*How the Parameter is Treated by the Broker*
priority	priority = **0**...**9** When there are several messages for a receiver that are awaiting delivery, higher priority messages (**5** through **9**) can move toward the front of the FIFO list. While there are circumstances where this is desirable, more often keeping a smooth FIFO flow is preferable.
timeToLive	timeToLive = <non-negative long integer value> Number of milliseconds added to the GMT time of the client when the message is produced to determine the **JMSExpiration** date-time of the message. If the *timeToLive* is **0**, the expiration date-time is also **0**, indicating the message is never to expire. The timeToLive feature ensures eventual delivery but can result in out-of-date deliverables when queues are not purged and when durable subscriptions are not formally unsubscribed.

Message Receivers, Listeners, and Selectors

MessageConsumer objects that are associated with a Topic do not automatically receive messages. Having an active session where an application subscribes to a topic does not result in the message being delivered to the application. You must use an asynchronous listener or a synchronous message receiver to ensure the message is delivered to an application.

Message Receiver

The receiver methods are synchronous calls to fetch messages. The different methods manage the potential block by either not waiting if there are no messages or timing out after a specified period.

Receive

To receive the next message produced for the consumer, use the `receive( )` method:

```
Message receive()
```

This call blocks indefinitely until a message is produced. When a `receive( )` method is called in a transacted session, the message remains with the consumer until the transaction commits. The return value is the next message produced for this consumer. If a session is closed while blocking, the return is `null`.

Receive with Timeout

To receive the next message within a specified time interval and cause a timeout when the interval has elapsed:

```
Message receive(long timeout)
```

where *timeout* is the timeout value in milliseconds.

This call blocks until either a message arrives or the timeout expires. The return value is the next message produced for this consumer, or `null` if one is not available.

Receive No Wait

To receive the next available message immediately or instantly timeout:

```
Message receiveNoWait()
```

The `receiveNoWait( )` method receives the next message if one is available. The return value is the next message produced for this consumer, or `null` if one is not available.

Note The `ReceiveNoWait( )` method is unlikely to provide effective message consumption in the Pub/Sub messaging model.The no-wait concept is useful for durable subscriptions, but is unlikely to produce results for normal subscriptions.
The method is very useful in the PTP messaging model where messages wait on a static queue.

Message Listeners

Invoke a message listener to initiate asynchronous monitoring of the session thread for consumer messages:

```
setMessageListener(MessageListener listener)
```

where *listener* is the message listener to associate with this session.

The listener is often assigned just after creating the destination consumer from the session, so the listener is bound to the destination where a consumer was just created. For example:

```
Sonic.Jms.MessageConsumer receiver =
                session.createConsumer(queue, String messageSelector);
receiver.setMessageListener(this);
```

and:

```
Sonic.Jms.MessageConsumer subscriber =
           session.createConsumer(topic, String messageSelector);
subscriber.setMessageListener(this);
```

As a result, asynchronous message receipt becomes exclusive for the session.

Note Message sending is not limited when message listeners are in use. Sending is always synchronous.

Message Selection

While some messaging applications expect to receive every message produced to a destination, other applications might want to receive only certain messages produced to a destination. The following techniques can reduce the flow of irrelevant messages to a message consumer:

- **Subscription to hierarchical name spaces (Pub/Sub)** — SonicMQ's hierarchical name spaces let subscribers point to content nodes (and, optionally, to sets of relevant subordinate nodes) to focus publishers into meaningful spaces. For more information, see Chapter 11, "Hierarchical Name Spaces."
- **Applying a message selector** — As shown in the preceding code examples, you can create consumers with a String parameter that holds a syntax that is a subset of SQL-92 conditional expressions. This SQL allows a consumer on a destination to filter and categorize messages in the message header and properties based on specified criteria.

Server-based or Client-based Topic Message Selectors

The default behavior of message selector filtering operations is defined by the messaging model:

- A queue receiver does its evaluation on the server as only one of the queue receivers will take the message instance.
- A topic subscriber is not receiving anything unique so it can take its subscribed messages to the client system and then select the messages that are acceptable.

However, there are cases where topic subscribers are particularly selective and the resources on the server far exceed the resources of the network and the clients. SonicMQ provides the option to perform subscription message selection on the server. A `setSelectorAtBroker(true)` method call on the connection factory before the topic connection is created enables this feature. See "Setting Server-based Message Selection" on page 87 for more information.

Scope of Message Selectors

Message selectors evaluate message header fields and properties. They do not access the message body. Although SQL supports arithmetic operations, message selectors do not. SQL comments are not supported.

A selector String greater than 1024 characters throws an exception.

Message Selector Syntax

A message selector is a `String` that is evaluated left to right within precedence level. You can use parentheses to change this order. A message selector string can contain combinations of the following elements to comprise an expression:

- **Literals and Indefinites** (See Table 16)
- **Operators and Expressions** (See Table 17)
- **Comparison tests** (See Table 18)
- **Parentheses** control the evaluation of an expression.
- **Whitespace** (spaces, horizontal tabs, form feeds, and line terminators) are evaluated in the same way as in C#.

For example, the following message selector might be set up on a `Bidders` topic to retrieve only high-priority quotes that are requesting a reply:

```
"Priority > 7 AND Form = 'Bid' AND Amount is NOT NULL"
```

Table 16. Literal and Identifier Syntax in Message Selectors

Selector	*Element*	*Format and Requirements*	*Constraints*	*Example*
Literals	String literals	Zero or more characters enclosed in single quotes.	None	`'sales'`

	Exact numeric literals	Numeric `long` integer values, signed or unsigned.	None	`57`
				`-957`
				`+62`

	Approximate numeric literals	Numeric `double` values in scientific notation.	None	`7E3` `-57.9E2`

		Numeric `double` values with a decimal, signed or unsigned.	None	`7.`
				`-95.7`
				`+6.2`

	Boolean literals	`true` or `false`	None	`true`

Table 16. Literal and Identifier Syntax in Message Selectors (*continued*)

Selector	*Element*	*Format and Requirements*	*Constraints*	*Example*
Identifiers	All	A case-sensitive character sequence.	Cannot be `null`, `true`, `false`, `NOT`, `AND`, `OR`, `BETWEEN`, `LIKE`, `IN`, or `IS`.	`JMSType`, `JMSXState`, `JMS_Links`, `PSC_Link`
	Message header field references	`JMSDeliveryMode`, `JMSPriority`, `JMSMessageID`, `JMSTimestamp`, `JMSCorrelationID`, or `JMSType`.	`JMSDelivery Mode`, and `JMSPriority` cannot be `null`.	`JMSType`
	JMSX-defined property references	`null` when a referenced property does not exist.	None	`JMSXState`
	SonicMQ-defined properties			`JMS_SonicMQ_ preserve Undelivered`
	Application-specific property names *(do not start with 'JMS')*			`Audit_Team`

Table 17. Operator and Expression Syntax in Message Selectors

Selector	*Element*	*Format and Requirements*	*Example*

Operators	Logical	In precedence order:
		`NOT`, `AND`, `OR`
			`a NOT IN ('a1','a2')`
			`a > 7 OR  b = true`
			`a > 7 AND b = true`

	Comparison	**=**, **>**, **>=**, **<**, **<=**, **<>** (for booleans and Strings: **=**, **<>**)	`a` **`>`** `7` `b` **`=`** `'Quote'`

	Arithmetic	In precedence order:
		● Unary + or -
		● Multiply ***** or divide **/**
		● Add **+** or subtract **-**
			`a >` **`+7`**
			`a` **`*`** `3`
			`a` **`-`** `3`

	Arithmetic range between two expressions	`id BETWEEN e2 AND e3` `id NOT BETWEEN e2 AND e3`	`a BETWEEN 3 AND 5` `a NOT BETWEEN 3 AND 5`
Expressions	Selector	Conditional expression that matches when it evaluates to `true`	`((4*3)=(2*6))= true`

	Arithmetic	Include:
		● Pure arithmetic expressions
		● Arithmetic operations
		● Identifiers with numeric values
		● Numeric literals
			`7*5`
			`a/b`
			`7`

	Conditional	Include:
		● Pure conditional expressions
		● Comparison operations
		● Logical operations
		● Identifiers with Boolean values
		● Boolean literals (`true`, `false`)
			`7>6`
			`a > 7 OR  b = true`
			`a = true`
			`true`

Table 18. Comparison Test Syntax in Message Selectors

Selector	*Element*	*Format and Requirements*	*Example*
Comparison tests	**IN**	`Identifier IN (str1, str2, ...)` `Identifier NOT IN (str1, str2, ...)`	`a IN ('AR','AP', 'GL')` `a NOT IN ('PR','IN', 'FA')`

	LIKE	`Identifier LIKE (str1, str2,...)`
		`Identifier NOT LIKE (str1, str2,...)`
		can be enhanced with pattern values:
		● Underscore (_) stands for any character
		● Percent (**%**) stands for any sequence of characters
		To explicitly defer the special characters _ and **%**, precede their entry with the Escape character.
			`a LIKE 'Fr%d'`
			is `true` for `'Fred' 'Frond'` and `false` for `'Fern'`
			
			`a LIKE '\_%' ESCAPE '\'`
			`true` for `'JMS_A'` and `false` for `'JMSPriority'`

	null	`Identifier IS NULL`
		`Identifier IS NOT NULL`
		for:
		● Header field value
		● Property value
		● Existence of a property
		Refer to SQL-92 semantics for more about comparisons that involve `null` values.
			`a is NULL`
			`a is NOT NULL`

Comparing Exact and Inexact Values

Comparing an `int` value and a `float` value is allowed.

Type conversion is as follows:

- Unary conversions are from `byte`, `short`, or `char` to a value of type `int` by a widening conversion and, otherwise, a unary numeric operand remains as is and is not converted.
- Binary conversions are called for by operands on data of numeric types. If either operand is of type `double`, the other is converted to `double`. If either operand is of type `float`, the other is converted to `float`. If either operand is of type `long`, the other is converted to `long`. Otherwise, both operands are converted to type `int`.

Steps in Listening, Receiving, and Consuming Messages

The following sections explain the steps required to receive and consume a Pub/Sub message within a connected session:

1. “Implement the Message Listener” on page 184.
2. “Create the Destination and Consumer, Then Listen” on page 185.
3. “Handle a Received Message” on page 185.
   a. “Get Message Header Fields” on page 185.
   b. “Get Message Properties” on page 186.
   c. “Consume the Message” on page 186.
   d. “Acknowledge the Message” on page 187.

Implement the Message Listener

Implement the standard message listener:

```
public class Chat: Sonic.Jms.MessageListener {
...
```

Create the Destination and Consumer, Then Listen

Once you obtain a `ConnectionFactory` object, you can use it to create a `Connection`. From the `Connection`, you can create a `Session`, and from the `Session` you can create a `MessageConsumer`.

To create a `MessageConsumer`, you call the `Session` object's `createConsumer( )` method. When you call this method, you pass in a `Destination` (either a `Queue` or `Topic`, both of which extend the `Destination` interface). If you pass in a `Queue`, the returned `MessageConsumer` acts in accordance with the PTP messaging model; if a `Topic`, the Pub/Sub messaging model.

After you create the `MessageConsumer`, you call its `setMessageListener( )` method, passing in the appropriate `MessageListener`. In the Chat sample, the `MessageListener` is the Chat object itself (`this`):

```
Sonic.Jms.Topic topic = subSession.createTopic("jms.samples.chat");
Sonic.Jms.MessageConsumer subscriber = subSession.createConsumer(topic);
subscriber.setMessageListener(this);
```

Handle a Received Message

In the following `Chat` sample code, the received message is assumed to be text and is output to the standard output stream:

```
public virtual void onMessage( Sonic.Jms.Message aMessage )
{
    Sonic.Jms.TextMessage textMessage = (Sonic.Jms.TextMessage) aMessage;
    String string = textMessage.getText();
    System.Console.Out.WriteLine( string );
}
```

When the received message type is uncertain, special message handling is required.

Get Message Header Fields

Use the `get( )` methods for the message header fields, such as:

```
getJMSMessageID()
```

Get Message Properties

Use the `get( )` methods for the data type of a property and then supply the property name and its value of the declared type. When a property requested does not exist in a message, the return value is `null`. Generically:

```
get[type]Property(String)
```

For example:

```
getIntProperty("OurInfo_AuditTrail")
```

Warning This example gets an **`int`** property that was set with (and stored as) a `long`. Attempting to get a property type that is not the type with which the property was set forces coercion of the value to the declared type. If the conversion is not valid, an exception is thrown. See Table 14, "Permitted Type Conversions for Message Properties."

Consume the Message

The application can pass the data in an accepted message to the business application for which it performs its services. Explicit acknowledgement of the message to the broker can be postponed until the business application acknowledges processing with a transaction or audit trail identifier. This value can be passed back to the producer if a reply is requested.

Acknowledge the Message

The acknowledgement mode is established when the session is created. Two of the acknowledgement modes are automatic: `AUTO_ACKNOWLEDGE` and `DUPS_OK_ACKNOWLEDGE`. Other acknowledgement modes require explicit invocation of the **`acknowledge( )`** method:

- If the mode for the session is `SINGLE_MESSAGE_ACKNOWLEDGE`, explicit acknowledgement acknowledges only the current message. Any messages not acknowledged are not released—thereby becoming available for redelivery—on the broker until the session ends.
- If the mode for the session is `CLIENT_ACKNOWLEDGE`, explicit acknowledgement acknowledges all messages previously received by the session.

Note The **`acknowledge`** method has no effect when the session is transacted or when the session acknowledgement mode is `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE`. See "Explicit Acknowledgement" on page 122 for more information.

Reply-to Mechanisms

The typical design pattern for request/reply is:

- Create a message you want to send
- Make a temporary destination
- Set the `JMSReplyTo` header to this destination
- Create a `MessageConsumer` on the destination
- Send the message
- Call `MessageConsumer.`**`receive`**`(timeout)` on the message

The **`JMSReplyTo`** message header field contains the destination where a reply to the current message should be sent. Messages with a `JMSReplyTo` value are typically expecting a response. If the `JMSReplyTo` value is `null`, no reply is expected. A response can be optional, and client code must handle the action. These messages are called **requests**.

A message sent in response to a request is called a **reply**. Message replies often use the `JMSCorrelationID` to ensure that replies synchronize with their requests. A `JMSCorrelationID` typically contains the `JMSMessageID` of the request.

Temporary Destinations Managed by a Requestor Helper Class

Under Pub/Sub, the `TopicRequestor` uses the session and topic that were instantiated from the session methods. The code snippets below are from the `TopicPubSub Requestor` and `Replier` samples. Notice that the code never actually manipulates the **`TemporaryTopic`** object; instead it uses the helper class `TopicRequestor`.

Requestor Application

The following code excerpt from the `TopicPubSub Requestor` sample application uses the helper class `TopicRequestor`:

```
Sonic.Jms.TopicRequestor requestor = new Sonic.Jms.TopicRequestor(session, topic);
Sonic.Jms.Message response = requestor.request(msg);
Sonic.Jms.TextMessage textMessage = (Sonic.Jms.TextMessage) response;
```

Replier Application

Synchronous requests leave the originator of a request waiting for a reply. To prevent a requestor from waiting, a well-designed application uses code similar to the following excerpts from the `TopicPubSub Replier` sample application:

1. Get the message:

```
public virtual void  onMessage(Sonic.Jms.Message aMessage)
    {
        try {
            // Cast the message as a text message.
          Sonic.Jms.TextMessage textMessage = (Sonic.Jms.TextMessage) aMessage;

            // This handler reads a single String from the
            // message and prints it to the standard output.
            try {
                System.String string_Renamed = textMessage.getText();
                System.Console.Out.WriteLine("[Request] " + string_Renamed);
```

2. Look for the header specifying **`JMSReplyTo`**:

```
Sonic.Jms.Topic replyTopic = (Sonic.Jms.Topic) aMessage.getJMSReplyTo();
if (replyTopic != null)...
```

3. Send a reply to the topic specified in **`JMSReplyTo`**:

```
Sonic.Jms.TextMessage reply =  session.createTextMessage();
```

Design for Handling Requests

The final steps taken by the message handler represent good programming style, but they are not required for message requests:

- Set the **JMSCorrelationID**, tying the response back to the original request.
- Use transacted session **commit** so the request is not received without the reply being sent, for example:

```
reply.setJMSCorrelationID(aMessage.getJMSMessageID());
replier.send(replyTopic, reply);
session.commit();
```

Producers and Consumers in Messaging Models

Table 19 lists a general messaging functionality that is consistent in both Publish and Subscribe and Point-to-point messaging.

Table 19. Producer and Consumer Common to Both Messaging Models

Interface	*Functionality in Messaging Models*
Destination extended by: **Queue**, **Topic**	● **Destination** supports concurrent use.
MessageProducer	● Can send message while connection is stopped. ● Close **MessageProducer** method. ● Supports message delivery modes **PERSISTENT**, **NON_PERSISTENT**, **NON_PERSISTENT_SYNC**, **NON_PERSISTENT_ASYNC**, and, for topics only, **DISCARDABLE**. ● Supports message **Time-to-Live**. ● Support message **priority**.

Table 19. Producer and Consumer Common to Both Messaging Models (*continued*)

Interface	*Functionality in Messaging Models*
MessageConsumer	• Close **`MessageConsumer`** method. • Supports **`MessageSelectors`**. • Supports synchronous delivery (**`receive`** method). • Supports asynchronous delivery (**`onMessage`** method). • Supports **`AUTO_ACKNOWLEDGE`** of messages. • Supports **`CLIENT_ACKNOWLEDGE`** of messages. • Supports **`DUPS_OK_ACKNOWLEDGE`** of messages. • Supports **`SINGLE_MESSAGE_ACKNOWLEDGE`** of messages.
Message extended by: **`TextMessage`** extended by **`XMLMessage`** **`MapMessage`** **`StreamMessage`** **`ObjectMessage`** **`BytesMessage`** **`MultipartMessage`**	• Message header fields. • Message properties. • Message acknowledgment. • Message selectors. • Access to message after being sent for reuse.

See Chapter 8, “Point-to-point Messaging” and Chapter 9, “Publish and Subscribe Messaging,” for programming concepts and distinguished functionality in each messaging model.

Chapter 8 Point-to-point Messaging

This chapter describes the Point-to-point (PTP) messaging model and explains how to use the features of that model. The chapter contains the following sections:

About Point-to-point Messaging

In the Point-to-point (PTP) messaging model, shown in Figure 17, a queue stores messages for as long as they are specified to live, waiting for a consumer.

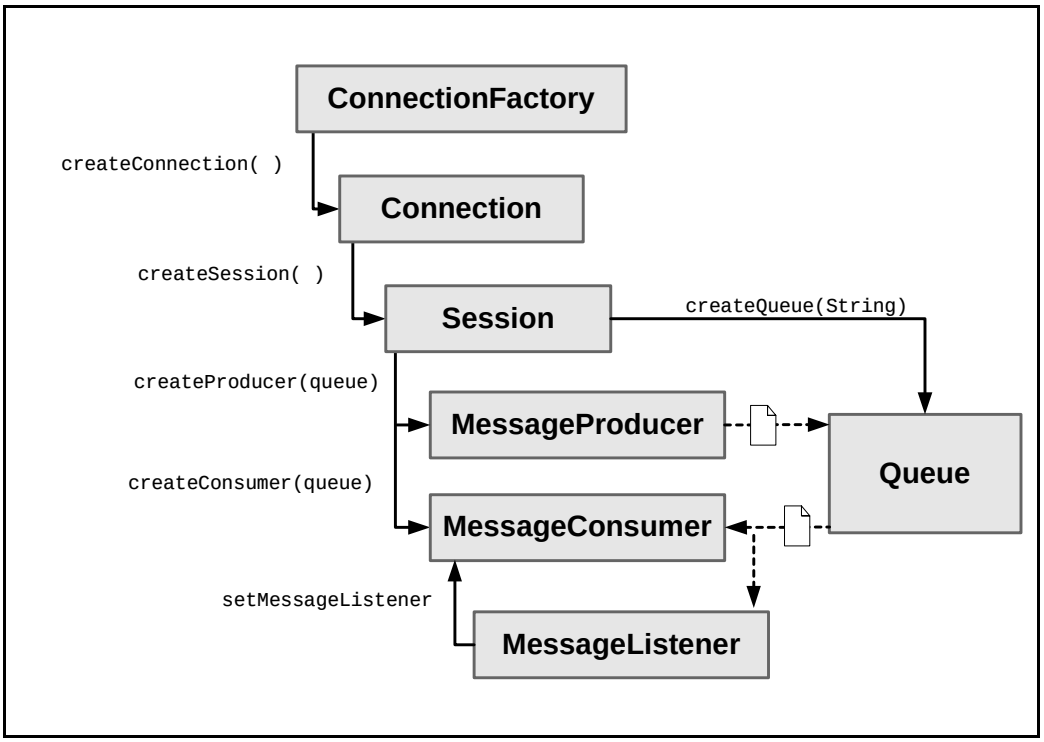


Figure 17. Point-to-point Messaging Model

The `QueueBrowser` class enables an application to browse the queue, examine its contents, and observe how message traffic is moving.

Queues must be created by the administrator before they can be used (except for temporary queues, which are created dynamically by users). See the *Progress SonicMQ Configuration and Management Guide* for information about maintaining queues.

The `QueuePTP` sample application, `Talk`, provides an example of how PTP applications are coded. The command that starts the `Talk` application specifies the sending queue and the receiving queue:

```
Talk -b broker:port -u user -p pwd -qs queue -qr queue
```

where:

- *`broker:port`* specifies the host on which the broker is running and the port on which it is listening.
- *`user`* and *`pwd`* are the unique user name and the user's password.
- `-qs` *`queue`* is the name of the queue for sending messages.
- `-qr` *`queue`* is the name of the queue for receiving messages.

Code Sample 4, from the `Talk` sample, shows how to create the objects used in PTP communication.

Code Sample 4. Talk Sample: Creating Objects for PTP

```
private void  talker(System.String broker, System.String username, System.String password,
                     System.String rQueue, System.String sQueue)
{
     // Create a connection.
     try
     {
          Sonic.Jms.ConnectionFactory factory;
          factory = (new Sonic.Jms.Cf.Impl.ConnectionFactory(broker));
          connect = factory.createConnection(username, password);
          sendSession = connect.createSession(false,
                                              Sonic.Jms.SessionMode.AUTO_ACKNOWLEDGE);
            receiveSession = connect.createSession(false,
                                              Sonic.Jms.SessionMode.AUTO_ACKNOWLEDGE);
     }
     catch (Sonic.Jms.JMSException jmse)
     {
          System.Console.Error.WriteLine("error: Cannot connect to Broker - " + broker);
          System.Console.WriteLine(jmse.StackTrace);
          System.Environment.Exit(1);
     }

     // Create Sender and Receiver 'Talk' queues
     try
     {
          if ((System.Object) sQueue != null) {
               Sonic.Jms.Queue sendQueue = sendSession.createQueue(sQueue);
               sender = sendSession.createProducer(sendQueue);
          }
          if ((System.Object) rQueue != null) {
               Sonic.Jms.Queue receiveQueue = receiveSession.createQueue(rQueue);
               Sonic.Jms.MessageConsumer qReceiver =
                         receiveSession.createConsumer(receiveQueue);
               qReceiver.setMessageListener(this);
               // Now that 'receive' setup is complete, start the Connection

               connect.start();
```

Message Ordering and Reliability in PTP

The PTP messaging model provides some unique features with respect to general message order and delivery.

Message Order

Queued delivery allows each message to be processed by one and only one message consumer. As a result, a series of messages might be consumed by several different message consumers, each taking a few messages.

Messages on a queue have characteristics that impact the ordering and reliability of messages:

- When a new message is put onto a queue with a higher **priority** set by the sender, an active message consumer takes the new message off the queue before taking an older message having a lower priority (provided that a message selector is not being used by the consumer).
- Queued messages that are not acknowledged are placed back on the queue (**reenqueued**) for delivery to the next qualified consumer. In the interim, a newer message might be received by a consumer.
- `MessageConsumer` objects have a **prefetch parameter** that retrieves a number of messages and caches them, for the client, for processing. If these messages are not processed by the client, they are returned to the queue.

Message Delivery

The following factors can impact the delivery of messages on a queue:

- Message selectors can limit the number of messages that a client receives. Messages can stay on the queue until a consumer provides either a suitable message selector or no message selector at all. A queue might appear empty to a consumer if none of the currently enqueued messages match the consumer's selection criteria.
- An administrator permanently disposes of queued messages (by clearing the queue).
- Message removal due to expiration might result in permanent disposal of a message or, if the message is flagged by the producer, the message being placed on the broker's DMQ. An administrative application can set up an authorized consumer on the DMQ to determine whether to recast the message, resend it as is, or discard it.

- Duplicate messages can be detected when transacted sessions are used if the broker is set up to manage the identifiers filed for a specified lifespan. With this broker setup, a commit to the specified identifier clears the index value, but any intervening sends that specify an already recorded identifier are rejected.

Note The effects of dynamic routing on message order and delivery are discussed at greater length in the *Progress SonicMQ Deployment Guide*.

Using Multiple MessageConsumers

Every `MessageConsumer` is prepared to receive the next available message on its associated queue. Since the PTP messaging model dictates one-to-one delivery semantics, each `MessageConsumer` only receives a subset of all the messages on a queue for which there are multiple active consumers. For example, a hundred messages on a queue for which there are four consumers might result in each consumer processing twenty-five messages each.

You can use either an asynchronous listener or a synchronous receiver for message delivery to an application for a queue. A synchronous consumer effectively generates a request for a message and wait for the message's delivery. With an asynchronous listener, implicit requests are generated for messages from the application's perspective, and the listener is invoked when a message is delivered.

Message Queue Listener

A message listener allows asynchronous processing of queue messages:

```
setMessageListener(MessageListener listener)
```

where *`listener`* is the message listener you want to associate with this session.

The listener is often assigned just after creating the consumer from the session:

```
Sonic.Jms.Queue receiveQueue = session.createQueue (rQueue);
Sonic.Jms.MessageConsumer qReceiver = session.createConsumer(receiveQueue);
qReceiver.setMessageListener(this);
```

As a result, asynchronous message receipt becomes exclusive for the consumer. Message sending is not limited when message listeners are in use. Sending is always synchronous unless you use the delivery mode `NON_PERSISTENT_ASYNC`, which results in asynchronous sending.

MessageConsumer

The `MessageConsumer` interface provides methods for synchronous calls to fetch messages. Variants allow for not waiting if there are no messages currently enqueued or for timing out after a specified wait period. These call methods are described in the following sections.

Receive

To synchronously receive the next message produced for the `MessageConsumer`, use the method:

```
Message receive( )
```

This call blocks indefinitely until a message is enqueued. When a `receive( )` method is called in a `transacted` session, the message remains with the `MessageConsumer` until the transaction is committed or rolled back. The return value is the next message delivered to this consumer. If a session is closed while blocking, the return value is `null`.

Receive with Timeout

To receive the next message on the queue within a specified time interval and cause a timeout when the interval has elapsed, use the method:

```
Message receive(long timeOut)
```

where *timeout* is the timeout value (in milliseconds).

This call blocks until a message arrives or the timeout expires, whichever occurs first. The return value is the next message delivered for this consumer, or `null` if one is not available.

Receive No Wait

To immediately receive the next available message on the queue or, otherwise, instantly timeout, use the method:

```
Message receiveNoWait( )
```

This call receives the next message if one is available. The return value is the next message delivered for this consumer, or `null` if one is not available.

Setting Prefetch Count and Threshold

SonicMQ extends the standard `MessageConsumer` interface, enabling you to set and get the following parameters of the message receiver for performance tuning:

- **PrefetchCount** — The number of messages that the consumer takes off the queue to buffer locally for consumption and acknowledgment (default value = 3).
- **PrefetchThreshold** — The minimum number of messages in the local buffer that will trigger a request for the delivery of more messages to the consumer. The number of requested messages is equal to the `PrefetchCount` (default value = 1).

For example, a `PrefetchThreshold` of **2** and a `PrefetchCount` of **5** causes a request to be sent to the broker for batches of five messages whenever the number of messages locally waiting for processing drops below two. The threshold value cannot be greater than the count value.

Use the following `set( )` and `get( )` methods for the prefetch count:

- **`setPrefetchCount`**:

  `Sonic.Jms.Ext.MessageConsumer.`**`setPrefetchCount`**`(int` *`count`*`)`

  where *`count`* is the number of messages to prefetch.

When the `PrefetchCount` value is greater than one, the broker can send multiple messages as part of a single `MessageConsumer` request, to improve performance.

- **`getPrefetchCount`**:

  `Sonic.Jms.Ext.MessageConsumer.`**`getPrefetchCount`**`()`

  Returns the `PrefetchCount` positive integer value.

Use the following `set( )` and `get( )` methods for the prefetch threshold.

- **`setPrefetchThreshold`**:

  `Sonic.Jms.Ext.MessageConsumer.`**`setPrefetchThreshold`**`(int` *`threshold`*`)`

  where *`threshold`* is the threshold value for prefetching messages.

  Setting this to a value greater than zero allows the `MessageConsumer` to always have messages available for processing locally, if any are available on the queue. This might improve performance.

  When the number of messages waiting to be processed by the `MessageConsumer` falls to (or below) the `PrefetchThreshold` number, a new batch of messages is fetched.

- **`getPrefetchThreshold`**:

  `Sonic.Jms.Ext.MessageConsumer.`**`getPrefetchThreshold`**`()`

  Returns the `PrefetchThreshold` positive integer value.

Browsing a Queue

A `QueueBrowser` enables a client to look at messages in a queue without removing them. Queue browsing in SonicMQ provides a dynamic view of a queue. As messages can be enqueued and/or dequeued very rapidly, browsing might not show every message on a queue over a given time interval. Browsing is very useful for assessing queue size and rates of growth. Instead of getting actual message data, you can also use the enumeration method to return just the integer count of messages on the queue.

Create the browser with a session method as follows:

```
Session.createBrowser(Queue queue)
```

where *queue* is the queue you want to browse.

A message selector string can be added to qualify the messages that are browsed. See "Message Selection" on page 178 for information about selector syntax. Create a browser with a message selector as follows:

```
Session.createBrowser(Queue queue, String messageSelector)
```

where:

- *queue* is the queue you want to browse
- *messageSelector* is the selector string that qualifies the messages you want to browse

Use the following methods to get browser information and close the browser:

- **getMessageSelector**:

  You can get the message selector expression being used with:

  ```
  String getMessageSelector()
  ```

- **getEnumeration**:

  You can get an enumeration for browsing the current queue messages in the sequence that messages would be received with:

  ```
  System.Collections.IEnumerator getEnumeration()
  ```

- **getQueue**:

  You can get the queue name associated with an active browser with:

  ```
  getQueue()
  ```

- **close**:

  Always close resources when they are no longer needed with `close()`

See "Browsing Clusterwide Queues" on page 214 for information about browsing clustered queues.

Handling Undelivered Messages

SonicMQ provides a service that a `MessageProducer` can request to handle undelivered messages. This service removes an undeliverable message from its queue, then re-enqueues the message on a special system queue. The message remains on this queue until acted on. This system queue, referred to as the dead message queue (DMQ), is usually managed by broker administrator applications. The name of this system queue is `SonicMQ.deadMessage`.

You can request to have undeliverable messages placed on the DMQ. You can set messages to:

- Be placed in the DMQ when the messages are found to be expired
- Be placed in the DMQ when a message cannot be delivered (for example, when the destination queue is not found)
- Request that a notification (an administrative event) be sent when the message is placed in the DMQ

Note There are several other reasons a message could be undelivered in a dynamic routing deployment. See *Progress SonicMQ Deployment Guide* for more about undelivered messages in the Dynamic Routing Architecture.

Setting Important Messages to be Saved if They Expire

Important messages should be sent with a PERSISTENT delivery mode and flagged to be preserved on expiration or when they cannot be routed successfully across routing nodes. You can choose to also generate an administrative notification when a message is enqueued on the DMQ. Code Sample 5 shows how to set messages for a PERSISTENT delivery mode to be preserved if undeliverable and to generate a notification if undeliverable.

Code Sample 5. Setting PERSISTENT Delivery Mode

```
// Create a BytesMessage for the payload.  Make sure the message
// is delivered within 2 hours (7,200,000 milliseconds).
// If expires, send a notification and save the message.
Sonic.Jms.BytesMessage msg = session.createBytesMessage();
msg.setBytes(payload);

// Set 'undelivered' behavior.
msg.setBooleanProperty(Sonic.Jms.Ext.Constants.PRESERVE_UNDELIVERED, true);
msg.setBooleanProperty(Sonic.Jms.Ext.Constants.NOTIFY_UNDELIVERED, true);

// Send the message with PERSISTENT, TimeToLive values.
qsender.send(msg,
             Sonic.Jms.DeliveryMode.PERSISTENT,
             Sonic.Jms.DefaultMessageProperties.DEFAULT_PRIORITY,
             7200000);
```

Setting Small Messages to Generate Administrative Notice

To determine whether delivery times are an issue, you can program an application to send a small message using high priority, with the expectation that this message will be delivered in ten minutes. Only notification events are needed. Code Sample 6 shows how to set messages to generate administrative notice.

Code Sample 6. Setting Messages to Generate Notification

```
// Create a BytesMessage for the payload.  Make sure the message
// is delivered within 10 minutes (600,000 milliseconds).
// If expires, send a notification.

Sonic.Jms.BytesMessage msg = session.createBytesMessage();
msg.setBytes(payload);

// Set 'undelivered' behavior. Using the property names that
// are defined as static const Strings in

// Sonic.Jms.Ext.Constants ensures catching errors.
msg.setBooleanProperty(Sonic.Jms.Ext.Constants.NOTIFY_UNDELIVERED, true);

// Send the message for fast delivery, or not at all.

qsender.send(msg,
             Sonic.Jms.DeliveryMode.NON_PERSISTENT,
             8,        // Expedite at a high priority
             600000);  // 10 minutes
```

In this example, when an administrative notification is received, you will know whether delivery times are large.

Life Cycle of a Guaranteed Message

A message is sent to the DMQ only when the application developer designates the message as a guaranteed message. The following sections explain the life cycle of a guaranteed message. See Chapter 10, "Guaranteeing Messages," for information about using the DMQ and guaranteeing messages.

Setting the Message to Be Preserved

You can choose to set the property of a message to declare that the entire message should be preserved if it is undeliverable as follows:

```
msg.setBooleanProperty(Sonic.Jms.Ext.Constants.PRESERVE_UNDELIVERED, true);
```

You can choose to also generate an administrative notification.

Setting the Message to Generate an Administrative Event

You can elect to be notified whether a message was delivered without needing to preserve the original message. This option is distinctly more efficient, both in terms of the message traffic density and the requirements of dequeueing undelivered messages. To declare that an administrative event should be generated, set the appropriate message property:

```
msg.setBooleanProperty(Sonic.Jms.Ext.Constants.NOTIFY_UNDELIVERED, true);
```

Sending the Message

The sending application sends the message metadata and the message payload. The application can expect the message to be delivered to an interested consumer.

Letting the Message Get Delivered or Expire

A message can be acknowledged as delivered to a consumer. A **`NON_PERSISTENT`** message is volatile in the event of a system outage, whereas a **`PERSISTENT`** message are restored in the event of a system outage.

Post-processing Expired Messages

When a message's expiration time (as marked in the message's `JMSExpiration` header field) has passed, the broker dequeues the message and examines the message producer's settings.

Dequeueing of expired messages only takes place when the enqueued messages are reviewed on the broker. Inert or low volume queues might have messages that expire but are not examined until a receive mechanism compels the broker to look at the message. Two properties are checked to see what processing steps are required:

- `JMS_SonicMQ_preserveUndelivered` — `If` **`true`**, the expired message is transferred to the DMQ
- `JMS_SonicMQ_notifyUndelivered` — If **`true`**, the expired message generates an administrative event

Processing Enqueued Expired Messages

When a message is transferred to the queue `SonicMQ.deadMessage`, the broker adds two properties:

- `JMS_SonicMQ_undeliveredReasonCode = reason code`
- `JMS_SonicMQ_undeliveredTimestamp = GMT time` [as long]

When a message is transferred to the DMQ due to expiration, it has the reason code **`UNDELIVERED_TTL_EXPIRED`**. The message retains its original `JMSDestination` header field value (unlike all other non-system queues, where the destination of each enqueued message matches the queue name).

Also, the message retains its original `JMSExpiration` header field value. When the message is retrieved from the DMQ, you can examine its properties including the time at which it was declared undeliverable, an indicator of the time on the system where the message expired.

Important Messages in the DMQ with a **`PERSISTENT`** delivery mode do not expire. If you have access to administrative functions on a broker, dequeue dead messages as soon as possible. Messages with **`NON_PERSISTENT`** delivery mode are volatile and perish if the broker restarts.

Sending Administrative Notification

When an expired message requests administrative notification, a notice is sent with the following information:

- **Undelivered Reason Code** — Stored in the `JMS_SonicMQ_undelivered_ReasonCode` property of the original message. For message expiration, the value of the reason code is **`UNDELIVERED_TTL_EXPIRED`** (which happens to be **`1`**). The message is undelivered because the message's `timeToLive` expired.
- **MessageID** — `JMSMessageID` of the original message.
- **Destination** — From `JMSDestination` of the original message.
- **Timestamp** — The time when the message was handled after a determination that it was undeliverable; also stored in the `JMS_SonicMQ_undeliveredTimestamp` property of the message if it is saved.
- **Broker Name** — The name of the broker where the notification originated. This information is important in clustered broker deployments.
- **Preserved** — As set in the `JMS_SonicMQ_preserveUndelivered` property of the original message. If **`true`**, the message was saved in the DMQ on the broker where the message was declared undeliverable.

Getting Messages Out of the Dead Message Queue

Code Sample 7 shows the use of synchronous receives for messages in the DMQ.

Code Sample 7. Getting Messages from the DMQ

```
// Create a MessageConsumer for the dead message queue.
Session session = connect.createSession(false,
                                        Sonic.Jms.SessionMode.CLIENT_ACKNOWLEDGE);
Queue dmq = session.createQueue ("SonicMQ.deadMessage");
MessageConsumer receiver = session.createConsumer(dmq);
connect.start();

// Empty the dead message queue.
while(true)
{
    Message m = receiver.receive();
    int code =
        m.getIntegerProperty(Sonic.Jms.Ext.Constants.UNDELIVERED_REASON_CODE);
    if (code == Sonic.Jms.Ext.Constants.UNDELIVERED_TTL_EXPIRED)
    {
        // Handle due to normal timeout.
        . . .
    }
}
```

Detecting Duplicate Messages

To avoid sending duplicate messages from two clients, or to avoid sending a duplicate message between client sessions, use a transacted `Sonic.Jms.Session` and use the `transactionID` parameter in the `commit( )` method to assign a 32-character (maximum size) unique universal identifier (UUID) to a message. The identifier might represent an audit trail value or a form identifier such as a purchase order number.

After you send messages in the transaction and then commit with your identifier, the commit throws an exception if the UUID is on file.

Use the `commit( )` method to commit all messages sent and received since the last commit or rollback. Use this method with a UUID:

```
commit(String transactionId, long lifespan)
```

where:

- *`transactionId`* is the UUID for duplicate transaction detection
- *`lifespan`* is the length of life of the UUID (in milliseconds)

If a transaction is rolled back because a duplicate UUID is detected, the following exception is thrown: `TransactionRolledBackException`.

Note Transactions using this commit feature are slower than normal transactions.

For more information about duplicate messages, see "Duplicate Message Detection Overview" on page 254.

Forwarding Messages Reliably

The `Sonic.Jms.Ext.Message` interface provides the `acknowledgeAndForward` method to reliably acknowledge a message received from a queue destination and forward the message to another queue destination.

To acknowledge a message and forward it to a new destination, use the method:

```
public abstract new void
    acknowledgeAndForward (Sonic.Jms.Destination destination,
                                                 int deliveryMode,
                                                 int priority,
                                                 long timeToLive)
```

where the variables are defined as follows:

- ***destination*** — The forwarding destination, a queue
- ***deliveryMode*** — The preferred delivery mode to use on the forwarded message (for example, `PERSISTENT` or `NON_PERSISTENT`)
- ***priority*** — The priority (0 - 9) to use on the forwarded message
- ***timeToLive*** — The new lifetime (in milliseconds) of the forwarded message

This method can be called only on messages that were received in a `Sonic.Jms.Ext.SessionMode.SINGLE_MESSAGE_ACKNOWLEDGE` session. The acknowledgment and the move to the new destination are performed as an atomic operation, guaranteeing that either both succeed or both fail. Other messages that might be received before this message, through the same session, are not affected.

You can use this method only for messages received from a queue destination and forwarded to a queue.

The optimal technique for routing messages to a remote queue is to build a transaction for a message wherein the message is not acknowledged to the broker queue from which it was received until it is securely enqueued in its target destination. SonicMQ provides a method that offers the increased reliability of acknowledge-and-forward without the overhead of a transacted session. Message moves assure that the body and property of the message are not disturbed by the action.

A message move requires that a client have the ability to both acknowledge receipt of a message and forward onto a new queue in a single, atomic action that couples the send method and the receipt acknowledge method. The session is required to use `SINGLE_MESSAGE_ACKNOWLEDGE`, which is the SonicMQ non-transacted extension of the `CLIENT_ACKNOWLEDGE` session parameter that is constrained to the current message only.

If the *priority*, *deliveryMode,* or *timeToLive* are not specified in the `acknowledgeAndForward( )` method, the values of those parameters are replicated from the original message. For example, the following method results in the use of the priority and delivery mode values of the message, while the interval between the timestamp and the expiration time is used as the time to live:

```
public abstract new void
     acknowledgeAndForward (Sonic.Jms.Destination destination)
```

The `acknowledgeAndForward( )` method does not acknowledge or forward previously received messages. If the method is called on acknowledged messages, an `IllegalStateException` is raised.

The definition of `acknowledgeAndForward` as **nontransacted** means that, while the commit is retained, no explicit rollback is available.

Note that topics are not currently supported for either message consumer destinations or message producer destinations under `acknowledgeAndForward`.

The following procedure describes how you can modify one of the sample applications to demonstrate the `acknowledgeAndForward( )` method.

◆ To modify a sample to show the acknowledgeAndForward behavior:

1. Copy the `ReliableTalk` sample source file, *net_client_install_dir*`/samples/TopicPubSub/ReliableTalk.cs`.
2. Change the line `textMessage.acknowledge()` to:

```
Sonic.Jms.Queue sendQueue = sendSession.createQueue(m_sQueue);
((Sonic.Jms.Ext.Message)textMessage).acknowledgeAndForward(sendQueue);
```

3. Change the line **`receiveSession = connect.createSession...`** to:

```
receiveSession =
connect.createSession(false,Sonic.Jms.Ext.SessionMode.SINGLE_MESSAGE_ACKNOWLEDGE);
```

4. Save the modified file and then compile it into a .exe file.
5. Run the modified `ReliableTalk` sample.

   Because of the change you made, the application automatically acknowledges any message it receives and forwards it to the send queue.

Dynamic Routing with PTP Messaging

The term **dynamic routing**, a concept familiar to network architects, is commonly used to describe the way routers talk to each other in order to maintain a list of connected routers. Sonic Dynamic Routing Architecture (DRA) is based on a similar concept. DRA is mostly managed in the communication layer, so programmers have minimal interaction with the physical deployment set up by the administrators, in the same way network applications that send an HTTP request to an IP address have no need to manage the routing of the request.

Fundamental to SonicMQ's reliable and secure message delivery are:

- Authentication in a SonicMQ node security domain
- Authorization for a destination maintained on the node

SonicMQ DRA provides active route optimization and accelerated acknowledge-and-forward transactional message forwarding while minimizing programmatic overhead.

Administrative Requirements

In Sonic dynamic routing deployments, an administrator must establish routing nodes and routing definitions, and must define users with routing ACLs. For dynamic routing of queue messages, an administrator must also establish global queues.

See the chapters "Configuring Routings" and "Managing SonicMQ Broker Activities" in the *Progress SonicMQ Configuration and Management Guide* for information about how to perform these administrative tasks.

Application Programming Requirements

To implement dynamic routing using PTP messaging, application programmers must send the queue messages with the destination format:

("*routing_node_name*::*global_queue_name*")

where the variables are defined as follows:

- ***routing_node_name*** — The name of an existing node (either a standalone broker or a cluster of brokers)
- ***global_queue_name*** — The name of an existing queue destination on that node that is set to be global

The *Progress SonicMQ Deployment Guide* provides examples of how you can implement dynamic routing in your applications. For detailed information about the different types of routing that SonicMQ provides, see the following:

- The "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* provides information about dynamic routing for queues in the PTP messaging model and dynamic routing for topics in the Pub/Sub messaging model.
- The "HTTP Direct Acceptors and Routings" chapter in the *Progress SonicMQ Deployment Guide* provides information about HTTP Direct routing.

Message Delivery with Dynamic Routing

Message behavior and handling when making use of dynamic routing is determined by several factors:

- What is the format of the destination name specified by the application?

  For example, the destination name can be specified in the following ways:

  - `destination` (non-remote destination)
  - `routing_node_name::destination` (remote destination)
  - `::destination` (global queue or topic on the current node)
- Is a broker a member of a cluster?
- Is the destination a queue or topic?
- If the destination is a queue:
  - Is the queue global?
  - Does the queue exist on a broker (either clustered or not)?
  - Is the queue a global queue elsewhere in the routing node?

Clusterwide Access to Queues

SonicMQ enables clusterwide access to queues, providing the following features for your applications:

- A client application with consumers, producers, and queue browsers can connect to any broker on the cluster and be able to receive from, browse, or send to any queue that has been administratively designated as clustered.
- Your applications can distribute messages on clustered queues.
- You can ensure Request/Reply, with a reply-to destination that is a temporary queue, with clustered queues.

Each broker in a cluster contains an instance of the clustered queue(s) configured for the cluster.

Sending to Clusterwide Queues

Sending to a clustered queue is similar to sending to a local queue on a broker. Each broker in the cluster contains an instance of the queue. When a message is sent to a broker, that broker places the message on its own instance of the queue. While a local clustered queue is accessible by specifying the destination as *queue_name* without any node syntax, a global clustered queue is accessible by specifying any of the three queue notations:

- *queue_name*
- *local_node*::*queue_name*
- ::*queue_name*

The direct interaction of producers with an instance of the queue on the local broker also means that transactions involving sending to clustered queues behave in the same way as transactions involving sending to local queues.

Receiving from Clusterwide Queues

From a consumer's perspective, receiving from a clustered queue on the local broker is no different than receiving from a local queue (see "MessageConsumer" on page 199). When a clustered queue on the local broker cannot satisfy the request for messages from its consumers, the clustered queue pulls messages from other clustered queue instances on neighbor brokers.

The clustered queue attempts to pull messages from corresponding neighbor brokers' clustered queue instances when:

- The clustered queue is empty and has consumers requesting messages.
- The clustered queue has more room and none of the requests can be satisfied by the existing messages on the queue.

Browsing Clusterwide Queues

A `QueueBrowser` created against a clustered queue has the same functionality as a `QueueBrowser` created against a non-clustered queue (see "Browsing a Queue" on page 201). The browsing of a clustered queue is an operation that examines the message content of the local broker's clustered queue instance only.

The following notes apply to a `QueueBrowser` for a clustered queue:

- A `QueueBrowser` for a clustered queue does not display messages that might be available for consumption on corresponding neighbor brokers' clustered queue instances.
- To browse the content of every instance of the clustered queue, you must connect to every broker in the cluster and create a queue browser each time.

Message Selectors with Clusterwide Queues

Message selectors are applied against messages in the local broker's clustered queue instance in the same manner as for a local queue (see "Message Selection" on page 178 and "Browsing a Queue" on page 201). A clustered queue instance on a broker receives messages from another neighbor broker's clustered queue instance if and only if a message satisfies at least one of the selectors in use by receivers connected to the broker.

In the event that a message consumer exists with no message selector, the neighbor broker does not have to take the time to evaluate the list of message selectors, as any messages will match the no selector case.

Clustered Queue Availability When Broker is Unavailable

If any broker in the cluster becomes unavailable as a result of software or hardware failure, all the messages on the clustered queue instances on that broker become unavailable until the broker is restarted. Since a clustered queue instance exists on all other brokers in the cluster, access for sending, receiving, and browsing continues uninterrupted for clients connected elsewhere in the cluster. However, the trapped messages are not available for browsing or receiving until the unavailable broker is restarted.

Note Clustered queues do not support the enforcement of strict message ordering.

Chapter 9 Publish and Subscribe Messaging

This chapter describes the Publish and Subscribe (Pub/Sub) messaging model and contains the following sections:

About Publish and Subscribe Messaging

The Publish and Subscribe (Pub/Sub) messaging model is shown in Figure 18. In Pub/Sub messaging, a message is sent to a **topic** and each consumer of that topic receives the message. This **one-to-many** model keeps topic producers (publishers) independent of topic consumers (subscribers). In fact, producers can send messages to topics where no consumers exist.

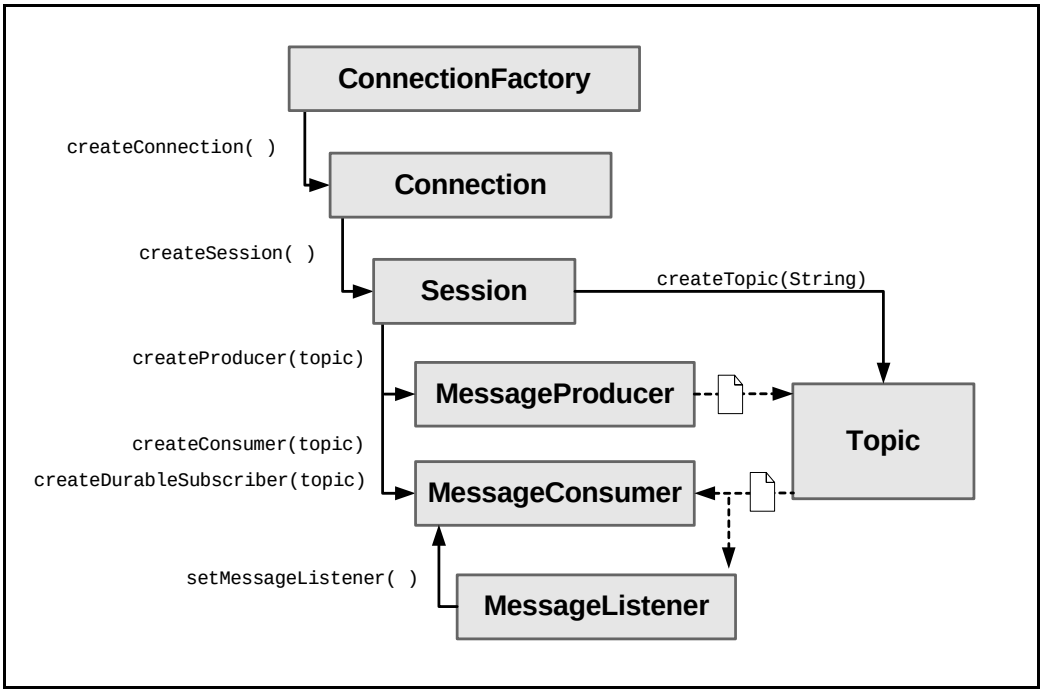


Figure 18. Publish and Subscribe Messaging Model

Mechanisms exist to allow messages to persist for consumers who have a durable subscription to a topic. The characteristics of durable subscriptions are discussed in "Durable Subscriptions" on page 224.

See Chapter 11, "Hierarchical Name Spaces," for information about how SonicMQ applications can subscribe to sets of topics.

Code Sample 8, from the `Chat` sample application, shows how to create the objects used in a `Session` for Pub/Sub communication: `Topic`, `MessageConsumer`, `MessageProducer`, and `Message`.

Code Sample 8. Creating Objects for Pub/Sub

```
//The topic is defined as a hierarchical topic
private const System.String APP_TOPIC = "jms.samples.chat";
...
     // Create Publisher and Subscriber to 'chat' topics
     try
     {
          Sonic.Jms.Topic topic = pubSession.createTopic(APP_TOPIC);
          Sonic.Jms.MessageConsumer subscriber = subSession.createConsumer(topic);
          subscriber.setMessageListener(this);
          publisher = pubSession.createProducer(topic);
          publisher.setDeliveryMode(Sonic.Jms.DeliveryMode.PERSISTENT);
          // Now that setup is complete, start the Connection
          connect.start();
     }
     catch (Sonic.Jms.JMSException jmse)
     {
          System.Console.WriteLine(jmse.StackTrace);
     }
...

     try
     {
          // Read all standard input and send it as a message.
          System.IO.Stream  stdin = System.Console.OpenStandardInput();
          System.IO.StreamReader stdinReader = new System.IO.StreamReader(stdin);
          System.Console.Out.WriteLine("\nEnter text messages to clients that subscribe to
                the " + APP_TOPIC + " topic." + "\nPress Enter to publish each message.\n");
          while (true) {
               System.String s = stdinReader.ReadLine();

               if ((System.Object) s == null)
                    exit();
               else if (s.Length > 0) {
                    Sonic.Jms.TextMessage msg = pubSession.createTextMessage();
                    msg.setText(username + ": " + s);
                    msg.setJMSDeliveryMode(Sonic.Jms.DeliveryMode.PERSISTENT);
                    publisher.send(msg);
               }
          }
     }
```

Message Ordering and Reliability in Pub/Sub

The Pub/Sub messaging model provides additional services to general message ordering and reliability, described in "Message Ordering and Reliability" on page 166.

General Services

Asynchronous message delivery allows messages be delivered with a range of options to assure an appropriate quality of service:

- The producer can set the message life span, delivery mode, and message priority.
- The broker stores the message for later delivery and manages both acknowledgement to the producer and acknowledgement from the consumer.
- The consumer can express a durable interest in a topic (durable subscriber).

While general services are impacted by many uncontrollable factors from latency to machine outages, there are internal factors that add complexity.

Reliable message delivery also deals with questions of ordering and redelivery.

Message Ordering

A predictable sequence of messages is a series of messages with the same priority from a single producer in a single session. Even if transacted, the messages are delivered sequentially from the broker to the consumers. The sequence of messages received by a consumer can be influenced by the following factors in the Pub/Sub messaging model:

- Changing a priority on a message from a producer can result in a delivery of a high priority message to a newly activated or reactivated subscription before an older message.
- Messages from other sessions and other connections are not required to be in specified sequence relative to messages from another session or connection.
- If a non-durable subscriber closes and then reconnects, it counts as a new subscriber. Order is only guaranteed within each connected session, not between sessions.
- Messages that are not acknowledged are redelivered to durable subscribers with an indication of the redelivery attempt. As a result, a redelivered message could be received after a message that was timestamped later.

- Durable subscriber disconnects and reconnects at a different broker. You can specify strict message order to ensure that messages will be received in the order they are sent, regardless of other factors that can affect that order. For information about message ordering with durable subscriptions, see "Message Order with Clusterwide Durable Subscriptions" on page 226.

Reliability

The assurance that a message is received by a consumer has several other influences in the Pub/Sub messaging model:

- A producer is never guaranteed that any consumer exists for a topic where messages are published.
- Consumer message selectors limit the number of messages that a client receives. Regular subscriptions and durable subscriptions with a message selector definition that excludes a message never receives that message.

Message destruction due to expiration or administrator action (removing a durable subscription) permanently disposes of stored messages.

Topics

Topics are objects that provide the producer, broker, and consumer with a destination. Topics can be predefined objects under administrative control, dynamic objects created as needed, or temporary objects created for very limited use. The topic name is a `String`, up to 256 characters.

SonicMQ provides extended topic management and security with **hierarchical name spaces**; for example, **`jms.samples.chat`**. Some characters and strings are reserved for the features of hierarchical topic structures:

- **`.`** (period) delimits hierarchical nodes.
- **`*`** (asterisk) and **`#`** (pound) are used as template characters.
- **`$`** (dollar sign) is used for internal topics (starting with $SYS or $ISB).
- **`:`** (colon) is used for dynamic routing.
- **`[[ ]]`** (double brackets) are used for shared subscriptions.

See Chapter 11, "Hierarchical Name Spaces," for more information.

Important Table 2, “Restricted Characters for Names” on page 83 lists characters that are not allowed in SonicMQ names. Refer to this list for restricted characters you must not use in your topic names.

MessageProducer (Publisher)

If you want your client application to send messages to a `Topic`, you must first create a `MessageProducer` in the session for the selected `Topic`. When you create a `MessageProducer` (via the `Session.createProducer( )` method), you can specify a default destination. If you specify a default destination, you do not have to specify a destination when you send a message.

You can also create a MessageProducer that is not bound to a default destination. You can do this by passing a `null` destination to the `createProducer( )` method. Then, to use the `MessageProducer` to send a message, you must explicitly call a form of the `send( )` method that specifies a valid destination. The following code creates a `MessageProducer` without a default `Topic`:

```
Sonic.Jms.MessageProducer publisher = session.createProducer(null);
Sonic.Jms.Topic topic = session.createTopic(“jms.sample.chat”);
publisher.send(topic, msg);
```

Creating the MessageProducer

This sample code creates a `MessageProducer` that specifies a default `Topic`:

```
Sonic.Jms.Topic topic = session.createTopic("jms.samples.chat");
publisher = session.createProducer(topic);
```

Creating the Message

The message is created using the `Session.createMessage( )` method for the preferred message type (for example, `Session.createTextMessage( )` creates a text message). The `Chat` sample application uses the following code to accept input and then create, populate, and send the input as a text message, prepended with the username of the `MessageProducer`:

```
while ( true )
{
    String s = stdin.readLine();
    if ( s == null )
        exit();
    else if ( s.length() > 0 )
    {
        Sonic.Jms.TextMessage msg = session.createTextMessage();
        msg.setText( username + ": " + s );
        publisher.send( msg );
    }
}
```

Sending Messages to a Topic

The `Chat` sample simply puts text into the body of the message and accepts every default that is provided for a message. The `send( )` method is:

```
publisher.send (Message message)
```

or

```
publisher.send (Message message,
                int deliveryMode,
                int priority,
                long timeToLive)
```

where:

- ***message*** is a `Sonic.Jms.Message`
- ***deliveryMode*** is [**NON_PERSISTENT|PERSISTENT|NON_PERSISTENT_SYNC|NON_PERSISTENT_ASYNC|DISCARDABLE**]
- ***priority*** is [**0...9**] where **0** is lowest and **9** is highest
- ***timeToLive*** is [**0...n**] where **0** is “forever” and any other positive value **n** is in milliseconds

MessageConsumer (Subscriber)

A `MessageConsumer` can subscribe to a topic. The `createConsumer( )` method, which creates a non-durable subscription, has the following parameters:

```
MessageConsumer createConsumer (Destination topic)
```

or

```
MessageConsumer createConsumer (Destination topic,
                                String messageSelector,
                                boolean noLocal)
```

where:

- ***topic*** is a `Topic` object you want to access
- ***messageSelector*** is a string that defines selection criteria
- ***noLocal*** is a boolean where **`true`** sets the option not to receive messages from subscribed topics published locally (by the same connection)

In a `Session`, multiple `MessageConsumer` objects can have overlapping subscriptions defined in their message selectors and hierarchical topics. In this case, all of the message consumers would get a copy of the message delivered.

Durable Subscriptions

A `MessageConsumer` can also express a durable interest in a topic (this is called a durable subscription). This means the `MessageConsumer` will receive all the messages published on a topic even when the client connection is not active. When the `MessageConsumer` expresses a durable interest in a topic, the broker ensures that all messages from the topic's publishers are retained until they either are acknowledged by the `MessageConsumer` or have expired. The **`createDurableSubscriber( )`** method has the following signatures:

```
TopicSubscriber createDurableSubscriber (Topic topic,
                                         String subscriptionName)
```

or

```
TopicSubscriber createDurableSubscriber (Topic topic,
                                         String subscriptionName,
                                         String messageSelector,
                                         boolean noLocal)
```

where:

- ***topic*** is a `Topic` object that specifies a topic to subscribe to
- ***subscriptionName*** is a string of arbitrary alphanumeric text. A subscription name is an identifier that allows a client to reconnect to a durable subscription.
- ***messageSelector*** is a string that defines selection criteria
- ***noLocal*** is a boolean. When set to **`true`**, the subscriber does not receive messages from subscribed topics published locally

The `TopicSubscriber` interface inherits from the `MessageConsumer` interface. It is recommended that you use the `MessageConsumer` interface instead; the `MessageConsumer` interface exposes all of the methods defined by the `TopicSubscriber` interface.

A subscription name is combined with the user name and the client identifier to define the durable interest. This construct creates many durable subscriptions that are easily understood and nonconflicting. The durable subscription identity is constructed from and indexed on:

- `username` — The username for authorization when logging on or for user identity
- `clientID` — The instance identifier in an application
- `subscription name` — The identity of the subscription within the application.

See Table 2, “Restricted Characters for Names” on page 83 for a list of restricted characters for durable subscriber names.

A durable subscription is not allowed for a temporary topic. An attempt to create a durable subscriber on a `TemporaryTopic` throws a `JMSException`.

While you can stop listening to a topic, there is broker overhead expended when trying to deliver messages to subscribers, especially when the messages might be persistent and the subscribers durable. The `Session` class’s `unsubscribe( )` method unsubscribes a durable subscription created by a client. This method deletes the state maintained on behalf of the subscriber by its message broker:

```
unsubscribe(String name)
```

where `name` is the name used to identify this subscription.

An **inactive durable subscription** is a durable subscription that exists but does not currently have a message consumer connected to it. A `MessageConsumer` must be inactive (closed) before using the `unsubscribe( )` method on that durable subscription.

An error occurs when a client tries to delete a durable subscription while the client has an active `MessageConsumer` for it.

Clusterwide Access to Durable Subscriptions

Messages in a durable subscription can be accessed from any broker in a cluster. A message that is published on one broker can be received by a client application that created a durable subscriber on any other broker in the cluster.

When a message is published for a disconnected durable subscriber, or a message is published while there is no active subscriber for the durable subscription on any broker in the cluster, that message is stored in the message store on the publishing broker. When the client application connects to any broker in the cluster and recreates the durable subscriber for the subscription, the messages stored earlier for that subscription are forwarded to the client application.

Message Order with Clusterwide Durable Subscriptions

If a client is publishing messages and the broker to which it is connected becomes unavailable, the client can reconnect to any other broker in the cluster and continue publishing messages. However, some of the messages published by the first session might be stored in the failed broker, and when that broker is restarted, they can be delivered out of order.

A similar situation can occur if an application publishes some messages, closes its session, then connects to a different broker in the cluster and continues publishing messages to the same topic. The strict order of messages delivered to the subscribers of the topic is not guaranteed across different publisher sessions.

With a single broker configuration, message ordering to durable subscribers is always guaranteed. In a clustered environment, SonicMQ supports optional strict message ordering to durable subscribers. This feature is optional in a clustered scenario because enforcing strict message order can lead to delays in delivery when messages intended for the durable subscriber are trapped on a crashed or partitioned broker. Applications that elect strict message ordering for durable subscriptions, therefore, must be able to tolerate delays in message delivery.

If an application is receiving messages from a durable subscription and the broker goes down or the application closes the current session, the application can later connect to any broker in the cluster and continue receiving messages from the durable subscription. In this situation, messages are received in the order they were sent even though the application started a new session (only if `setDurableMessageOrder` was enabled).

Strict message ordering is not enabled by default. An application can select strict message order enforcement in the `ConnectionFactory` or in the topic session. The setting at the topic session always takes precedence over any settings at the `ConnectionFactory`. You can enable preservation of message order for reconnecting durable subscribers as follows:

- When set in the connection factory, all durable subscribers created on one of the connections are created with message ordering enforced:

```
Sonic.Jms.Cf.Impl.ConnectionFactory.setDurableMessageOrder
                                                    (boolean durableSubscriberMessageOrder)
```

- When set in the session, all durable subscribers created on the session use this value, which overrides the value set in the connection factory:

```
Sonic.Jms.Ext.Session.setDurableMessageOrder (boolean durableSubscriberMessageOrder)
```

It is possible to change the message ordering of a durable subscriber each time it connects. However, once the durable connects with message ordering disabled, there is no guarantee how long it will take to restore message order after reconnecting it with message ordering enabled. It is possible to have messages out of order in this case.

Availability of Clusterwide Durable Subscription After Reconnecting

This section explains the availability of clusterwide durable subscriptions after a broker becomes unavailable and then reconnects. If a broker goes down as a result of a software or hardware failure, all messages stored on behalf of durable subscriptions on that broker become unavailable until the broker is restarted. When a client application creates a durable subscriber, that client receives the messages from the brokers in the cluster that are up and running, but cannot receive messages stored on the broker that went down, until it is restarted.

If the broker to which the client is connected goes down and the client reconnects to another broker in the cluster, it is possible that some messages unacknowledged by the application in the previous session will remain in-doubt until the broker that went down is restarted. In this case, if strict durable message order is requested, the client might not be able to receive a subscriptions's messages until that broker is restarted.

A client application can publish messages as long it is connected to some broker in the cluster even if other brokers in the cluster are down.

Figure 19 shows an example of three brokers in a cluster. In this example, broker **B** becomes unavailable, then reconnects. Message delivery proceeds differently depending on whether durable message ordering is enabled, as explained in the following sections.

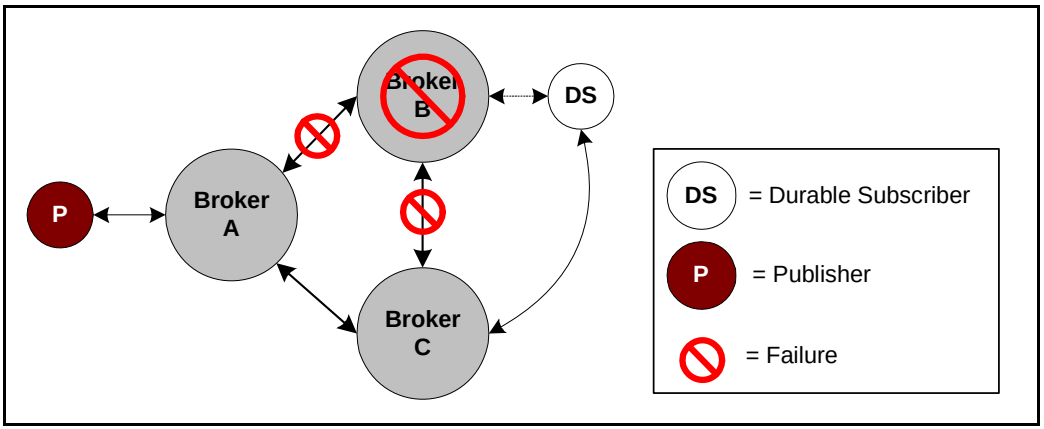


Figure 19. Clusterwide Durable Subscription Availability After Failure

Durable Message Ordering Enabled

In Figure 19, broker **B** is unavailable and durable subscriber **DS** moves from broker **B** to broker **C**. In this case durable message ordering is enabled and, as a result, delivery of messages to **DS** are delayed until broker **A**'s connection to **B** is restored. This delay is due to the following:

- To preserve message order for the **DS**, messages sent to **B** destined for **DS** are stored on **B** until the connection between **B** and **C** is restored.
- Any messages stored on **B** from publisher **P** must be delivered to **DS** (which is now on **C**) before new messages from publisher **P** can be delivered.

Therefore, message delivery cannot continue until broker **B** comes back online. The messages stored **A** on cannot be sent to **C** until the connection is restored between brokers **A** and **B** and the in-doubt message state between brokers **A** and **C** is resolved, or there is a risk of redelivery. This example shows how, with message ordering enabled, it is possible for message delivery to be delayed when the broker on which the durable was active becomes unavailable.

Durable Message Ordering Disabled

When strict durable message ordering is not enabled in the example shown in Figure 19, message delivery is not delayed for the durable subscriber **DS**. Messages stored on broker **B** are not delivered to **DS** on **C** until the connection is restored. Messages in-doubt between **A** and **B** are not delivered until the connection between them is restored and the doubt is resolved, but new messages from publisher **P** are sent to **DS** on **C**. Once the brokers reconnect to **B**, the skipped messages are delivered out of order.

Dynamic Routing with Pub/Sub Messaging

Dynamic routing, a concept familiar to network architects, defines the way routers talk to each other to maintain a list of connected routers. The Sonic Dynamic Routing Architecture (DRA) is based on the same concepts. Most of the DRA complexity is managed in the communication layer so programmers have minimal interface with the architecture implemented by the administrators, in the same way network applications that send an HTTP request to an IP address do not have to manage the routing of the request.

Fundamental to SonicMQ's reliable and secure message delivery are:

- Authentication in a SonicMQ node security domain
- Authorization on a destination maintained on the node

This static design can result in a high messaging volume on some brokers. While load balancing and clustering can force clients to try other connections, those solutions can be time-intensive and the result is a static list of connections instead of a static connection.

SonicMQ DRA provides remote publishing and subscribing for topic messages. This feature allows applications to publish messages to remote nodes, and enables subscribers to receive messages published from remote nodes.

Note There are two ways to use dynamic routing with Pub/Sub messaging: with global subscription rules or with remote publishing. See the *Progress SonicMQ Deployment Guide*.

Administrative Requirements

In all cases of dynamic routing and remote publishing, an administrator must establish routing nodes and routing definitions, and must define users with routing ACLs. Remote subscribing requires the administrator to establish subscription rules for each remote subscriber.

See the “Configuring Routings” and “Managing SonicMQ Broker Activities” chapters in the *Progress SonicMQ Configuration and Management Guide*.

Application Programming Requirements

To implement remote publishing of topic messages, the application programmer must publish topic messages with the destination format: ***routing_node_name*::*topic_name***

Use this syntax when you want to deliver the message only to the subscribers of a single remote node. If your application messages are to be delivered according to the global subscription rules set up administratively, you should use the ***topic_name*** syntax.

It is also possible to administratively connect the topic spaces of two nodes so that messages published to a topic on one node are delivered to subscribers on the other node without using a special destination format. This technique is called **global subscriptions**.

Global subscriptions where the topic spans two nodes is implemented entirely as an administrative task. Programmers do not have to be aware that they are sending to, or receiving from, topics that cross routing nodes.

The *Progress SonicMQ Deployment Guide* provides examples of how you can implement dynamic routing or remote publishing in your applications. For detailed information about the different types of routing that SonicMQ provides, see the following:

- The "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* provides information about dynamic routing for queues in the point-to-point domain and dynamic routing for topics in the publish and subscribe domain.
- The "HTTP Direct Acceptors and Routings" chapter in the *Progress SonicMQ Deployment Guide* provides information about HTTP Direct routing.

Message Delivery with Remote Publishing

Message behavior and handling with remote publishing is determined based on how the destination name is referenced when the application creates the destination. For example, the destination name can be referenced as:

- *`destination`* (non-remote)
- *`routing_node_name`*`::`*`destination`* (topic on *`routing_node_name`*)
- `::`*`destination`* (topic on the current node)

Shared Subscriptions

A problem in topic subscriptions is that there are often cases where one application acting as a topic subscriber cannot process messages as fast as messages are being published. This leads to a bottleneck, where the subscribing application falls further and further behind.

Two typical solutions are:

- **Forward messages to queue** — A single application consumes messages from the topic and forwards the messages to a queue. The shortcomings of this approach are:
  - The need to guarantee the operation of the forwarding application, which must be transacted to guarantee delivery
  - The extra hops required for the topic/queue forwarding

- **Multiple Standard Subscribers** — By creating multiple subscribers, each subscriber receives every message and application logic must either serialize requests against a common resource such as a persistent storage mechanism, or check with a central controlling program to resolve the duplicates.

SonicMQ provides a solution to the bottleneck problem by enabling you to establish groups of topic subscribers that share subscriptions to allocate the message load between them. Within these groups, members share the subscription so that while every message is delivered to the group, each message is delivered to, and consumed by, only one member of the group. These group members can be located on dispersed computers over diverse connections. The implementation is compatible with clusters of brokers so that the members of a consumer group can connect to different brokers in a SonicMQ cluster. Regular subscribers, durable subscribers, and participants in a shared subscription can be active concurrently on a broker.

Figure 20 shows an example where the clients, including those within a group, might be connected to different brokers. In this example, the publishers are producing to one topic and all the subscribers are actively consuming on that topic. Using shared subscriptions within the two subscriber groups provides the following performance:

- Consumer 1 and Consumer 2 receive every message only once.
- Group 1 and Group 2 receive every message only once. The members of the group each receive a subset of the complete set of messages.

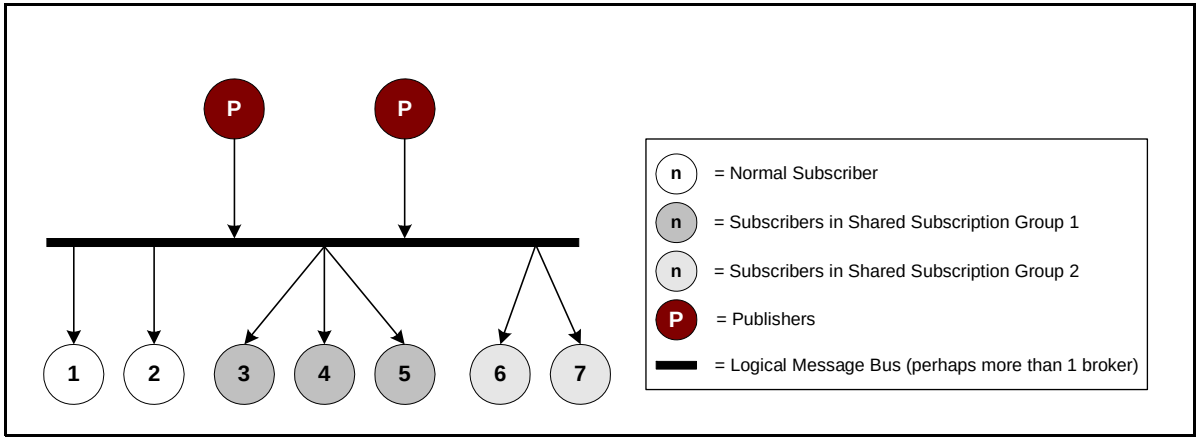


Figure 20. Illustration of Subscribers Abstracted from Specific Broker Connections

Table 20 shows how messages are received in the shared subscription configuration shown in Figure 20. In the scenario shown in Table 20, ten sequential messages are sent. The **X**'s in the table indicate which subscribers receive each message. **Subscriber 6** fails after the sixth message is received, and all subsequent messages are delivered to the remaining member of that shared subscription group.

Table 20. Example of Messages Received Under Load Balancing

Message #	*Normal Subscribers*		*Shared Subscription Group 1*			*Shared Subscription Group 2*	
	Sub 1	**Sub 2**	**Sub 3**	**Sub 4**	**Sub 5**	**Sub 6**	**Sub 7**
1	X	X	X			X	
2	X	X		X			X
3	X	X			X	X	
4	X	X	X				X
5	X	X		X		X	
6	X	X			X		X
Subscriber 6 Fails							
7	X	X	X				X
8	X	X		X			X
9	X	X			X		X
10	X	X	X				X

Note that the failure of **Subscriber 6** does not cause messages delivered to it to be reallocated to **Subscriber 7**. Also, if **Subscriber 6** is not durable, **Message 7** might not get to **Subscriber 7**.

Features of Using Shared Subscriptions in Your Applications

Implementing shared subscriptions with groups of topic subscribers provides the following features:

- You can create groups of subscribers that share subscriptions so that each message is delivered to only a single member of the group.
- Members of the group can be run on multiple processors or computers. Group members do not have to know about or be able to communicate with to each other (except so far as they are related through messaging).
- Members of a shared subscription group can connect to different brokers in a cluster.
- Shared subscriptions can be used in non-durable, high-throughput/low-latency applications, providing a solution to the problem of slow applications leading to flow-control in situations where non-persistent/non-durable subscribers are normally used.
- Applications using shared subscriptions use the standard C# API.
- Messages are allocated evenly to members of a shared subscription group. However, clients that are slow to the point of flow-control are explicitly skipped in message allocation. In addition, delivery to local subscribers (on the same broker as the publisher) are favored. When publishers and subscribers are co-located on one broker, or when the subscriber is on a different broker than the publisher, the following conditions may apply:
  - The order that messages are allocated to group members may vary between subsequent cycles throughout the group.
  - There is no guarantee that some members might not be allocated messages more than once in some cycles throughout the group.
  - Fairness is determined as a long-term average, rather than a short-term strict round-robin. Table 21 shows an acceptable variation for **Subscriber Group 1** shown in Figure 20.

Table 21. Balanced and Fair Delivery to a Shared Subscription Group

Message #	*Shared Subscription Group 1*		
	Subscriber 3	*Subscriber 4*	*Subscriber 5*
1	X		
2		X	
3			X
4			X
5		X	
6	X		
7	X		
8		X	
9		X	
10			X
11	X		
12			X

- If group members are non-durable, or if messages are DISCARDABLE, then messages might be lost due to broker or client failure. For **non-durable subscribers**, this implies:
  - Client failures might cause the loss of all messages delivered to that client (but unacknowledged). This is true for all delivery modes including `PERSISTENT`.
  - Failure of a broker may cause the loss of all undelivered and unacknowledged messages held on that broker.

Usage Scenarios for Shared Subscriptions

The following sections describe cases where implementing shared subscriptions can improve the performance of your applications. Figure 21 illustrates these scenarios.

Pure Load-balancing

The throughput in Pub/Sub messaging is effectively limited to the speed of the slowest topic subscriber. To divide the slowest application across two computers, you can have two identical topic subscribers acting as a single consumer.

Effectively, the goal is to have a shared subscription group act similarly to a single subscriber with similar durability and acknowledgement modes.

Fault Resilience

This scenario guarantees that a topic subscriber application receives messages exactly once, and is resilient to both broker and application failures.

In this example, subscribers **`g1`** and **`g2`** are in a shared subscription group. The message stream from publishers **`P1`** and **`P2`** is divided between them (with indirect routing from broker **`A`** to brokers **`B`** and **`C`**). This configuration continues to process messages even if any of the following components fail:

- Broker **`B`**
- Broker **`C`**
- Topic subscriber **`g1`**
- Topic subscriber **`g2`**

The normal subscriber (**`1`**) receives all messages.

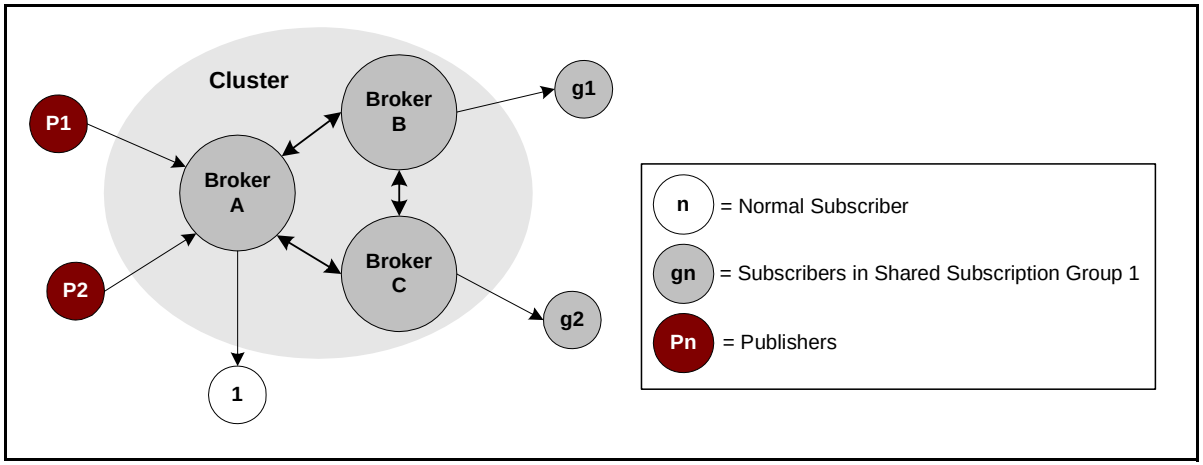


Figure 21. Fault Resistance Across Shared Subscription Topic Subscribers on a Cluster

Note This configuration is **fault resistant** rather than fault tolerant because this is not a message replication scheme. A failure of broker **B**, for example, might still cause messages to be trapped or lost on broker **B** (as group members are non-durable). What will happen is that subsequently published new messages will be automatically redirected entirely to topic subscriber **g2**.

Highly-Variable Processing Times

There are situations where a topic subscriber is fast enough to handle the message flow, but where some individual messages take significant processing. In a typical messaging application, the processing of messages following such a message must wait.

An example of this is an application where the topic subscriber creates a conversation with a particular publisher (perhaps to request more information, or to satisfy a business transaction). This is a common scenario in many financial applications where a request to buy might involve creating an order, and sending conformation information back to the sending client.

In the example shown in Figure 22, publishers **P1** and **P2** are publishing these requests. The normal subscriber **1** is simply listening to every message and, perhaps, recording it in an audit log. Shared subscription topic subscribers **g1** and **g2** are listening for orders.

When **P1** sends a request, it might be handled by **g1**. In this example, this action involves a long duration conversation back with **P1**. Without the availability of **g2** as a shared subscription topic subscriber, **P2** cannot also send a request until **g1** finishes. In this configuration, then, implementing shared subscriptions for the group of subscribers increases the efficiency of the processing.

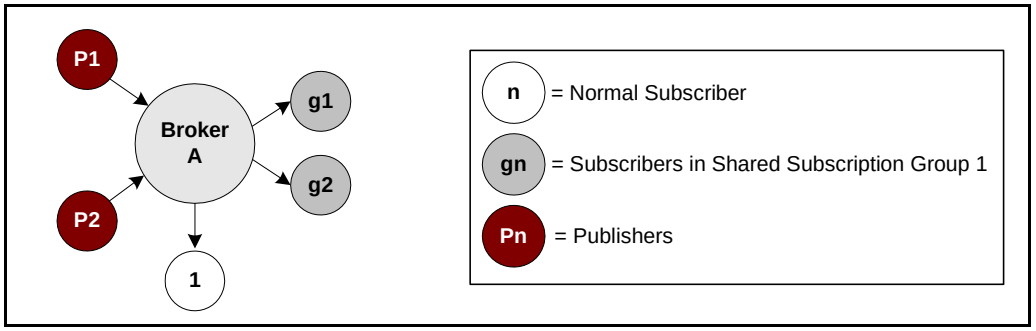


Figure 22. Application in a Shared Subscription Group Processes Messages Once at Most

Defining Shared Subscription Topic Subscribers

Use the following syntax to name topic subscribers within a group:

```
[[prefixName]]topicName
```

A `JMSException` is thrown if the group name is invalid.

Once a topic is created, a subscription can be created on that topic, with the following additional requirements:

- The following call creates a topic object:

```
Sonic.Jms.Session.createTopic(String name)
```

  The topic name must meet these requirements:

  - If the name starts with **[[** then it must:
    - contain matching closing characters **]]**
    - contain some characters after the closing brackets **]]**

- The group prefix (between **[[** and **]]**) can be any Unicode character string up to 64 characters. The following characters are not allowed in a prefix name:
  - **$** (as ANY character)
  - **** (backslash)
  - ***** (asterisk)
  - **#** (pound)
  - **.** (period)
  - **::** (double colon)
  - **[** (open bracket)
  - **]** (close bracket)
- Using **[[** and **::** are invalid in the same topic name (in any order)

- The following methods creates a subscriber on topic `T` as part of the group (`T` is defined as `[[prefix]]topic`):

```
Session.createConsumer(Topic T)
Session.createConsumer(Topic T, boolean nolocal, String selector)
```

  - Members of the group are those with:
    - the identical group name
    - the identical topic name
    - `nolocal` may be `true` for group subscribers.
  - Access control is based on the destination, without the group name. That is, for `[[prefix]]topic`, only the `topic` part is checked in the authorization policy.

Important Selectors need not match, but if selectors are used and mismatched, you should use broker-side selectors.

- The following methods create a durable subscriber on topic `T`, as part of the group:

```
Sonic.Jms.Ext.Session.createDurableSubscriber  (Topic T, String
                                                subscriptionName);
Sonic.Jms.Ext.Session.createDurableSubscriber  (Topic T, String
                    subscriptionName, boolean nolocal, String selector)
```

  - Members of the group are those with:
    - the identical group prefix (case sensitive)
    - the identical topic name (case sensitive)

    For example, a message published to **`T.A`** is delivered to one member of each of the following groups:
    - **`[[group]]T.A`** — Only one subscriber in the cluster receives the message
    - **`[[grp2]]T.A`** — Only one subscriber in the cluster receives the message
    - **`[[group]]T.*`** — Only one subscriber in the cluster receives the message

Note

- Group **`[[g]]T.A`** is not part of **`[[g]]T.*`** — Because there is no overlapping based on wildcards
- Group **`[[G]]T.*`** is not related to **`[[g]]t.*`** — Because group names are case sensitive

  - Selectors need not match, but if selectors are used and mismatched, you should use broker-side selectors.
  - Durable subscribers in the same group must follow normal durable subscriber rules. That is, the members of the group must differ in one or more of the following:
    - subscription name
    - Client ID
    - User Name
  - `nolocal` might be `true` for group durable subscribers, but locally published messages might not be delivered to another group member
- You cannot publish to a topic that has a group prefix.
- You should not use a topic that has a group prefix `ReplyTo` in its name because you cannot publish to it:

```
Sonic.Jms.Message.setJMSReplyTo(Topic T)
```

Message Delivery to a Broker with Shared Subscriptions

This section describes message delivery in both a single broker and cluster configurations.

Single Broker Behavior with Shared Subscriptions

In the example shown in Figure 23, a topic has both normal and shared subscription topic subscribers. Two publishers, **P1** and **P2**, are connected to broker **A**. There are two normal topic subscribers, **1** and **2**, and a group of shared subscription topic subscribers **g1**, **g2**, and **g3**.

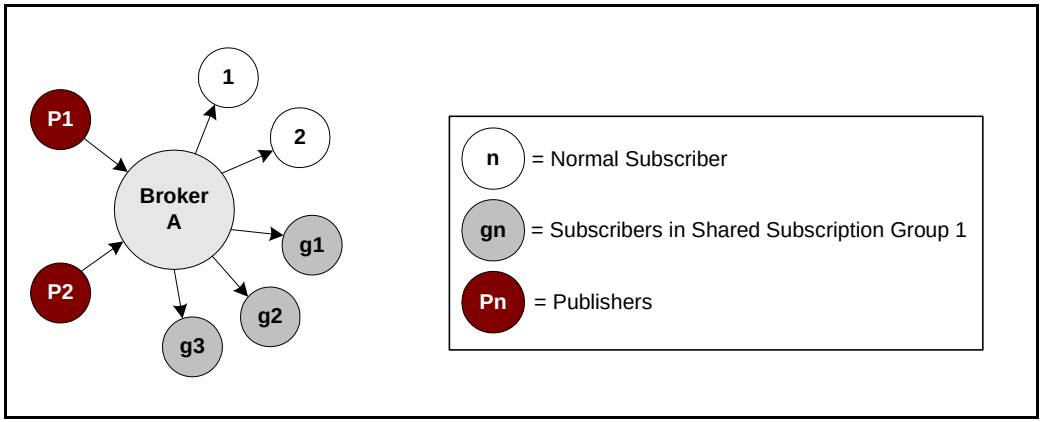


Figure 23. Shared Subscriptions with a Single Broker

In SonicMQ messaging, when a message arrives at a broker from a publisher, it must be delivered to:

- Each connected normal subscriber, on whatever broker the subscriber is connected to.
- Each disconnected normal durable subscriber, on whatever broker the subscriber is connected to.
- One member of each shared subscription group.

In this example, when broker **A** receives a message from publisher **P1**, the broker must deliver a copy of that message to all normal subscribers whose subscription matches the message topic and properties.

For each shared subscription group, however, the decision is slightly more complex because only one group member must receive (and acknowledge) the message. This behavior calls for the broker to allocate delivery between members of the group, attempting to deliver the message to a member. If space is not available on one member subscriber, then the next group member is tried.

Connecting Group Member

When a new subscriber is added to the group, the new member receives the next published message, not the first unprocessed message. Unprocessed messages are not reallocated. Once a message is sent to a particular client context, it is not reallocated unless the client fails.

Similarly, adding a new group member breaks a flow control situation. Existing clients that are flow controlled continue to be blocked until the subscriber that caused the flow control situation either processes messages or is closed. However, a new publisher is not flow controlled because the new publisher is allocated to the new group member.

Selectors and Shared Subscriptions

Members in a shared subscription group are allowed to have different selectors. The broker checks the selector before allocating messages to a group member, but only if the user has asked for the use of broker-side selectors.

If you want to use selectors on shared subscriptions, you should either:

- Make sure all selectors are the same
- Use broker-side selectors

Failure to do so might cause messages to go to clients that discard them when they apply the selector locally.

Disconnecting Group Member (Non-durable)

When a member of the shared subscription group is closed, or disconnects, all messages allocated to the client might be discarded if the subscription is not durable. This means that both `NON_PERSISTENT` and `PERSISTENT` messages can be discarded when the subscription is not durable.

No attempt is made to reallocate unacknowledged messages allocated or delivered to the client. In addition, pending messages allocated to a particular client are lost, even if they are not delivered to the message listener's `onMessage( )` method.

Shared Durable Subscriptions

Shared subscription durable subscribers act individually. The group allows one message to be allocated to a single member of the group. After the allocation occurs, the durable subscription behaves normally. If the durable subscription disconnects, unacknowledged messages go into the persistent storage mechanism, and are not delivered until the durable subscriber reconnects.

An advantage to using shared subscriptions is that if any member of a group is connected, that member receives new messages. The only time a broker allocates messages to a group of disconnected durable subscriptions is when:

- No connected subscriptions are known
- All connected subscribers are flow controlled

Once a message goes to a disconnected durable subscription, it can be retrieved only by that subscription when it next is connected. There is no sharing or reallocating of disconnected durable messages.

Cluster Behavior with Shared Subscriptions

SonicMQ supports shared subscriptions in a cluster. In the example shown in Figure 24, a topic has both normal and shared subscription topic subscribers. Two publishers, **P1** and **P2**, are connected to separate brokers **A** and **C**. Normal subscribers **1** and **2** are also distributed through the cluster. A group of shared subscription topic subscribers (**g1**, **g2**, **g3**, and **g4**) are also connected throughout the cluster.

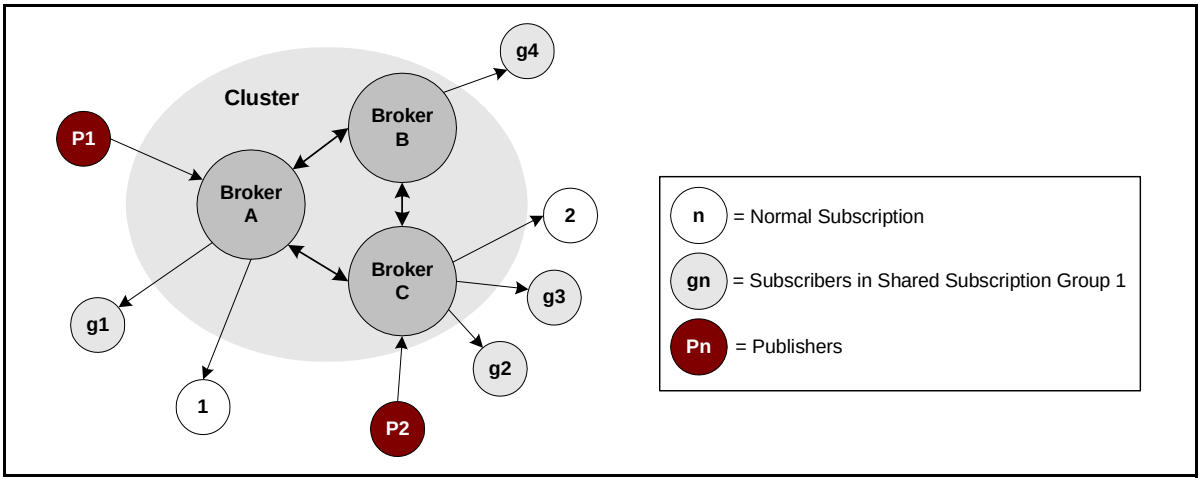


Figure 24. Cluster with Shared Subscriptions

In this example, a message is published by **P1** to a topic on broker **A**. Broker **A** handles the message for normal subscribers as follows:

- Delivers the message to local subscribers, in this case, subscriber **1**
- Sends a copy to other brokers with normal subscribers, in this case, subscriber **2** on broker **C**

Broker **A** must also decide to which group subscriptions the message is targeted. For each group, the broker must decide whether to handle the message locally, or to push the message to another broker. To make these decisions, the broker maintains a list of brokers that have shared subscribers in a group. For each new message, the broker uses this list to decide whether to handle a message locally, or to forward it to another broker. Preference is given to local subscribers. If a local subscriber cannot accept a message, that message is sent to the next subscriber in the cluster.

When a broker receives a message over an interbroker connection with a list of group subscriptions, the receiving broker takes responsibility for message delivery. The receiving broker tries to deliver the message to shared subscription local subscribers. If all locally connected subscribers are closed or are full, the receiving broker must either:

- Discard the message, if no broker is known to have active subscribers
- Forward the message to another broker where subscribers exist

Messages are not forwarded in a loop. At most, message delivery is attempted on every broker. The last broker always accepts the message at:

- Connected local subscribers (even if flow controlled)
- Disconnected durable subscribers

If no subscribers exist, the message is discarded.

Shared Subscriptions and Flow Control

When a broker gives a message to a member of a group subscription, the broker chooses the member as follows:

1. The broker searches for any local subscribers connected to the broker that are free to immediately process the message, without causing persistent storage mechanism I/O or causing a flow control situation. This means that the subscriber's in-memory buffer on the broker has plenty of space for the message. If such subscribers are found, the broker gives the message to the next such subscriber.
2. If the broker cannot find a subscriber in its initial search, the broker checks to see whether it can give the message to another broker in the same cluster—a broker that has members for the same group subscription. If so, the broker attempts to give the message to the other broker, allowing the other broker to complete the process of choosing a subscriber.
3. If the broker cannot successfully give the message to another broker in the same cluster, it once again searches for local connected subscribers. This time, however, the broker searches for subscribers that have flow-to-disk functionality enabled. If such subscribers are found, it gives the message to the next such subscriber. Although this causes persistent storage mechanism I/O, it nevertheless delivers the message to a connected subscriber.

4. If the broker cannot find subscribers with flow-to-disk enabled, the broker looks for disconnected durable subscribers. If such subscribers are found, it gives the message to the next such subscriber. Again, this causes persistent storage mechanism I/O, but it avoids a flow-control situation and its attendant blocking of message producers.
5. If the broker cannot find any disconnected durable subscribers, it gives the message to the next available subscriber, even if it means causing a flow-control situation.

Once the broker allocates a message to a particular group member, the normal SonicMQ mechanisms apply. For example:

- If the subscriber is non-durable, and it closes, unacknowledged messages are lost.
- If the subscriber is durable, and it closes, unacknowledged messages are stored in a persistent storage mechanism.
- If the subscriber does a `rollback( )` on a transacted session, or if it does a `Session.recover( )`, messages are redelivered to that same subscriber.
- If the subscriber has client-side selectors, messages delivered to it might be discarded if the selector does not match.

Interactions with Shared Subscriptions

The following sections describe some examples of interactions that can occur with shared subscriptions.

Session Recovery

When the `recover( )` method is called on a session, message delivery is stopped in the session, then restarted with the oldest unacknowledged message for that session. The redelivered messages have their `JMSRedelivered` flag set to `true`—a setting that can be made only by the broker.

No reallocation of messages occurs from one shared subscription topic subscriber to another.

Transacted Sessions

A transacted session delays acknowledgement of messages received on it until the `commit( )` method is called. For non-durable subscriptions, closing or failing the session causes those unacknowledged messages to be discarded.

For durable subscriptions, closing or crashing the session causes the messages originally delivered to the subscriber to be stored in the persistent storage mechanism.

The behavior of shared subscription topic subscribers is similar to normal subscribers in failure situations. That is, messages are not acknowledged, and are discarded.

Similarly, if the `rollback( )` method is called on a transacted session, the normal SonicMQ behavior is followed. That is, message delivery is restarted with the unacknowledged messages in the transaction. These messages are redelivered to the same client session and are not reallocated to different members of the group.

Transacted sessions provide no additional protection from message loss for these non-durable shared subscription topic subscribers. No reallocation of messages occurs from one load-balanced subscriber to another (except when the subscription is durable).

Shared Subscriptions with Remote Publishing and Subscribing

SonicMQ dynamic routing architecture (DRA) includes:

- **Remote Publishing** — Allows a client to publish to a remote node.
- **Global Subscriptions** — Allows an administrator to define a rule that allows a subscription on one node to be propagated to another node.

Shared subscriptions are intended to work with both remote publishing and global subscriptions. For remote publishing, the interaction is minimal. A remote node publishing (using the syntax "*node_name*::*topic_name*") acts like a local publisher. That is, messages published on one node directed to a second node go to normal subscribers, and round-robin to shared subscribers.

See "Dynamic Routing with Pub/Sub Messaging" on page 229 and the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* for information about remote publishing and global subscriptions.

For global subscriptions, there are two points to note:

- Remote subscriptions are only valid for topics without the group prefix. That is, you can have a rule that propagates `T.A`, but not `[[g]]T.A`.
- The subscribing node can have multiple group subscribers, but these act like a single subscriber when triggering a rule. That is, if two subscribers create subscriptions to `[[g]]T.A`, this acts (for global subscription purposes) as a subscription to `T.A`. Adding a new member to the group should not fire a new rule. Secondly, the remote subscription goes away when the last member of the group unsubscribes and closes.

Figure 25 illustrates the behavior of remote publishing and global subscriptions with shared subscriptions. This example includes the two routing nodes **SingleNode** and **ClusteredNode**:

- **SingleNode** — A single broker routing node containing broker **C**
- **ClusteredNode** — A multi-broker routing node containing brokers **A** and **B**

Publishers **P1** and **P2** publish messages on **ClusteredNode**, and **SingleNode** contains shared subscriber groups **g** and **G**.

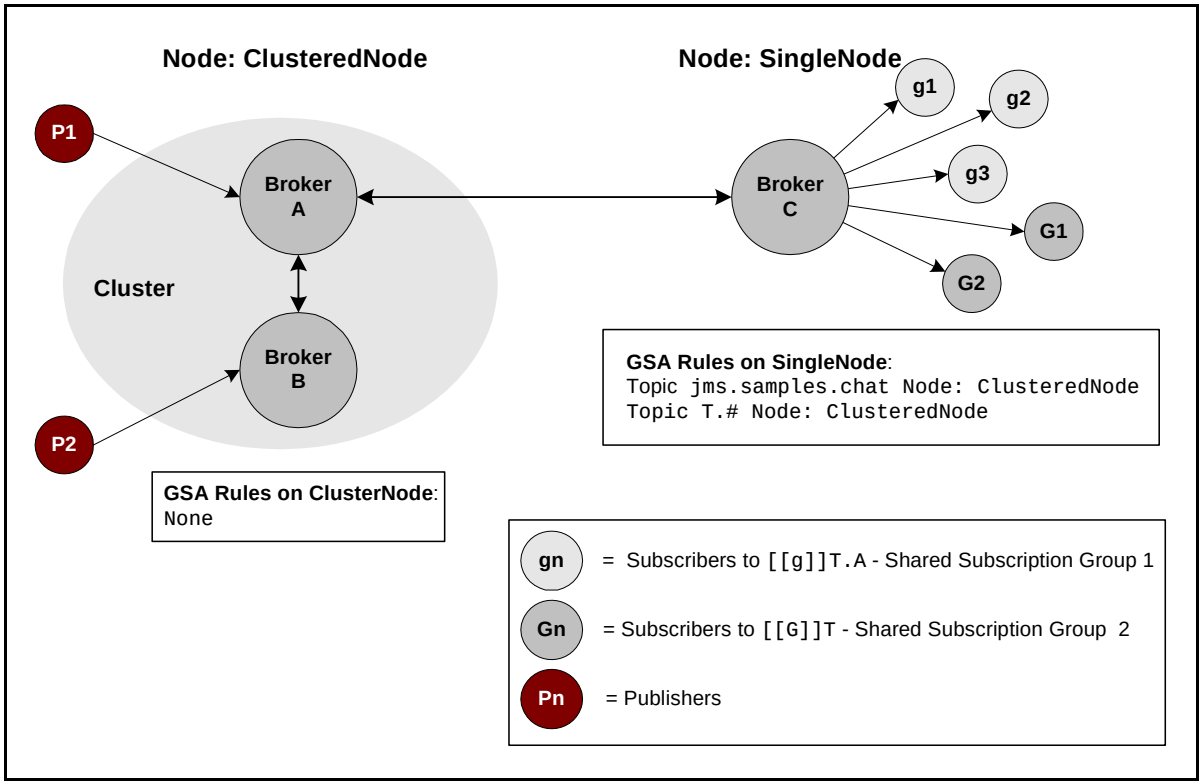


Figure 25. Clustered Node with Publishers and Single Node with Shared Subscriber Groups

The routing node **SingleNode** has the following rules and subscribers:

- Rules on **SingleNode**:
  - Topic pattern `T.#` **->** propagated to `ClusteredNode`
  - Topic pattern `jms.sample.chat` **->** propagated to `ClusteredNode`
- Subscribers on **SingleNode**:
  - Three subscribers in the shared subscription group **g**: **`[[g]]T.A`**
  - Two subscribers in the shared subscription group **G**: **`[[G]]T`**
  - One non-shared subscriber to **ClusteredNode**

The following sections describe the message routing behavior of this configuration for different scenarios using remote publishing and global subscriptions with shared subscription groups.

Example of Global Subscriptions

In this example using global subscriptions, a rule is defined that allows a subscription sent to a broker on one node, **SingleNode**, to be propagated to another node, **ClusteredNode** (see Figure 25). Messages sent by publishers on broker **A** are routed as follows:

- Publishers on broker **A** publish to **ClusteredNode** — The message goes to the correct subscriber on **SingleNode**.
- Publishers on broker **A** publish four times to **T.A** — The messages alternate between **[[g]]T.A** subscribers on **SingleNode**.

Example of Global Subscriptions with a Cluster

This example shows how an existing broker connection (in this case, on broker **A**) is shared by another broker (broker **B**) in the same cluster when publishing (see Figure 25). Messages sent by publishers on broker **B** do the following:

- Publishers on broker **B** publish to **ClusteredNode** — The message goes to the correct subscriber on **SingleNode**.
- Publishers on broker **B** publish four times to **T.A** — The messages alternate between **[[g]]T.A** subscribers on **SingleNode**.

Example Without Global Subscriptions

If there are no global subscriptions in the configuration shown in Figure 25, the messages sent by publishers on **ClusteredNode** are handled as follows:

- Publish four times to **T** — The messages go nowhere on **SingleNode**.

Example of Global Subscription Maintenance

This example shows how global subscriptions are maintained when individual subscribers on a node become unavailable. This example refers to the configuration shown in Figure 25.

- Stop one **[[g]]T.A** subscriber on **SingleNode** and publish four times to **T.A** — The messages alternate between the remaining two **[[g]]T.A** subscribers on **SingleNode**.
- Stop another **[[g]]T.A** subscriber on **SingleNode** and publish four times to **T.A —** The messages go to the last remaining **[[g]]T.A** subscribers on **SingleNode**.
- Stop the last **[[g]]T.A** subscriber on **SingleNode** and publish four times to **T.A** — the messages are not sent to **SingleNode**.

Example of Remote Publishing

This example describes how remote publishing allows a client to publish to a remote node, in this example, **SingleNode**, as shown in Figure 25. Messages sent by publishers on broker **A** are routed as follows:

- Publishers on broker **A** publish four times to **SingleNode::T.A** — The messages alternate between **[[g]]T.A** subscribers on **SingleNode**.
- Publishers on broker **A** publish two times to **SingleNode::T** — The messages alternate between **[[G]]T** subscribers on **SingleNode**.

Example of Remote Publishing with a Cluster

This example shows how an existing broker connection (in this case, on broker **A**) is shared by another broker (broker **B**) in the same cluster when publishing (see Figure 25). Messages sent by publishers on broker **B** do the following:

- Publishers on broker **B** publish four times to **SingleNode::T.A** — The messages alternate between **[[g]]T.A** subscribers on **SingleNode**.
- Publishers on broker **B** publish two times to **SingleNode::T** — The messages alternate between **[[G]]T** subscribers on **SingleNode**.

Chapter 10 Guaranteeing Messages

This chapter provides information about preventing duplicate messages and guaranteeing message delivery. The first part of this chapter explains how you can detect duplicate messages and prevent messages from being delivered more than once. The second part of the chapter provides information about how you can use the SonicMQ Dead Message Queue (DMQ) features to guarantee that messages are not discarded until a client processes them. The chapter contains the following sections:

Introduction

SonicMQ can guarantee message delivery when the broker system to which a client connects determines that:

- Messages are not duplicates of ones already delivered.
- Messages that are undeliverable can be channeled into a holding area for administrative handling.

Duplicate Message Detection Overview

In some applications, it is critical that multiple messages with identical content not be sent. When a message is successfully enqueued on a SonicMQ message broker, there is no possibility that it will be duplicated. The duplicate message detection feature handles problems that might arise:

- If there is a connection or network failure between the client and the message broker, the application might commit the send of a message, but the acknowledgment of the commit might be lost due to network failure. The client, in this case, doesn't know the message was sent.
- The application might fail after the commit occurs. Even though the message was sent and committed at the SonicMQ level, the application has no persistent record of this and might try to resend the message when it is restarted. Using XA connections (and a transaction manager) can also alleviate this situation.
- If the application is not properly designed for concurrent operation, two instances or threads in the application might try to send the same or similar message.

SonicMQ Extensions to Prevent Duplicate Messages

SonicMQ allows a commit to take an optional index parameter and possibly a lifespan parameter as follows:

```
Session.commit(transactionID, lifespan)
```

where:

- *`transactionID`* is a universally unique identifier generated by the application that is guaranteed to be unique
- *`lifespan`* is the duration for which the `transactionID` is intended to be saved (in milliseconds)

The indexed commit operation is supported on SonicMQ transacted queue sessions and topic sessions. It functions as shown in Table 22.

Table 22. Session.commit Behavior

Condition	*Action*
`transactionID` exists	Throw an exception.
`transactionID` does not exist	Store a new value of `transactionID` and continue with normal `Session.commit( )` behavior.

Support for Detecting Duplicate Messages

The SonicMQ message broker stores the index for duplicate detection in a persistent storage mechanism. The mechanism is always created when storage is initialized. The mechanism name is created from the `BrokerName` by default, but you can also explicitly specify this name.

Message brokers in a cluster can share one persistent storage mechanism. Different brokers in a cluster can point to the same persistent storage mechanism by assigning them the same value for `IndexedTxnTableName`. You can set this value in the Sonic Management Console in the **Broker/Properties/Storage** dialog box in the **Table Name** field. See the *Progress SonicMQ Configuration and Management Guide* for information about setting broker properties in the Sonic Management Console.

Note that every indexed commit requires a persistent storage mechanism action, so sessions using duplicate message properties are significantly slower than other sessions. These persistent storage mechanism actions are sequentially committed, so if two clients use the same ID at the same time, only one is successful. There is no window where the two clients both succeed.

Dead Message Queue Overview

Messages that expire or are viewed by SonicMQ as undeliverable are called dead messages. Dead messages can occur in multi-node deployments, which are discussed in the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide*.

SonicMQ provides the Dead Message Queue (DMQ) to handle dead messages. Both topic and queue messages can be sent to the DMQ. In PTP messaging, undeliverable and unexpired queue messages go to the DMQ, while in Pub/Sub messaging only undeliverable topic messages go to the DMQ. Also, in some cases where HTTP Direct messaging is used, messages might go to the DMQ. Your application can either request to receive notifications of undeliverable or expired messages, or include methods to handle these messages on the DMQ.

When you implement the SonicMQ dead message features, the SonicMQ broker can find messages that exceed their time to live (TTL) and should expire, or that cannot be routed due to some external network error. In this situation, the broker can either or both:

- Save the message in a dead message queue (DMQ)
- Generate an administrative notification (management event)

Your application can listen for administrative notifications, browse the DMQ, and deal with undelivered messages as appropriate for your application. The following sections explain how you can adapt your applications to handle dead messages.

Note Messages sent with a `NON_PERSISTENT` delivery mode are subject to a lower quality of service than `PERSISTENT` messages. `NON_PERSISTENT` messages in the DMQ are not retained after a planned or unplanned shutdown of the broker. These messages must be processed in the same broker session in which they occur, otherwise they are discarded.

When a network failure occurs while both brokers are still running, messages sent with a `NON_PERSISTENT` delivery mode can be lost. If one routing node sends a `NON_PERSISTENT` message to another node and the network fails, additional messages are blocked at the originating broker pending a reestablishment of the connection. However, an indoubt message sent with a `NON_PERSISTENT` delivery mode might be lost.

Topic messages published with a `DISCARDABLE` delivery mode that are undeliverable are not saved in the DMQ, and do not generate notifications. `DISCARDABLE` topic messages are lost when undeliverable, even if the `JMS_SonicMQ_preserveUndelivered` property is set.

What Is an Undeliverable Message?

In the case of broker-to-broker routing across routing nodes, there are cases where messages are considered undeliverable. (See "Dynamic Routing with PTP Messaging" on page 211, "Dynamic Routing with Pub/Sub Messaging" on page 229, and the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* for information about Progress Sonic's Dynamic Routing Architecture.) These cases include the following types of messages:

- **Unroutable messages** — Queue or topic messages that arrive at a routing queue where the information on the routing is missing or incomplete.
- **Indoubt messages** — Queue or topic messages that are forwarded to another routing node, but where the handshaking needed to ensure once-and-only-once delivery of messages is interrupted due to network or hardware failure and cannot be re-established within the configurable `RoutingIndoubtTimeout`.
- **Expired messages** — Queue messages that do not progress during routing for a configured period of time, specified by the time to live on the message or routing timeout.

There are other reasons why a message might not be delivered, including timeouts and network failures. See the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* for descriptions of various scenarios under which messages are not delivered. See "Undelivered Message Reason Codes" on page 267 for reason codes and descriptions of errors indicating undelivered messages.

Messages that do not make forward progress during routing for a configured period of time are transferred to the DMQ. This period of time is specified by the TTL parameter of the send method.

Using the Dead Message Queue

If your application specifies the DMQ option for each message, all expired or undeliverable messages are sent to the DMQ. The DMQ is treated like a normal queue in that it can be browsed or read using normal objects (`QueueBrowser` and `QueueReceiver`). The only special handling feature of these queues is that messages are not allowed to expire from them.

The DMQ is created and populated by SonicMQ, and has the following properties:

- Is created automatically by SonicMQ (all running SonicMQ brokers have an active DMQ)
- Is always named: `SonicMQ.deadMessage`
- Is a simple queue (neither clustered nor global)
- Cannot be deleted

As with other queues, messages that have a `JMSDeliveryMode` of `NON_PERSISTENT` are not available in the DMQ after a system shutdown (either planned or unplanned).

Guaranteeing Delivery

When you use the DMQ, any expired or undeliverable message is guaranteed to be preserved on the broker. To ensure that expired or undeliverable messages are preserved, you must configure your application to:

- Request that expired or undelivered messages be preserved.
- Monitor the DMQs.
- Handle all messages that arrive in the DMQ.

Enabling Dead Message Queue Features

You enable the DMQ features only on a message-by-message basis. You must specifically request enqueuing and notifications of administrative events, or the DMQ is not used. Enabling the DMQ in this way prevents the DMQ from accidentally filling up and shutting down the broker.

See "JMS_SonicMQ Message Properties Used for DMQ" on page 261 for information about the message properties that request enqueuing on the DMQ.

Monitoring Dead Message Queues

It is very important that your application monitor the DMQs and deal with messages that arrive there. When any of these system queues exceeds its maximum queue size, the broker is shut down.

The SonicMQ broker will shut down if the DMQ exceeds its configured capacity. Prior to shutting down the broker, however, the DMQ raises an administrative event when it exceeds a fraction of its maximum size. This notification factor is set by default to 85%. You can reset this value in the broker's **Properties** dialog box using the Sonic Management Console. See the "Configuring Queues" chapter in the *Progress SonicMQ Configuration and Management Guide* for information about resetting this value.

Warning Applications should not directly add messages to the DMQ by creating `QueueSenders`. Recommended access to the DMQ is through `QueueBrowsers` and `QueueReceivers`.

Note Queue messages that are enqueued in the DMQ retain their original `JMSDestination` and `JMSExpiration` values. The destination value of a topic message on the DMQ changes to include the node name to which it was routed. For example, the destination might be changed from "`MyTopic`" to "`NodeA::MyTopic`" to reflect the node to which the message was undeliverable.

Ensure that `QueueBrowsers` and `QueueReceivers` on the DMQ check the `(Sonic.Jms.Message).m.getJMSDestination()` for the original topic or queue.

Default DMQ Properties

By default, SonicMQ creates the DMQ with the properties listed in Table 23.

Table 23. Dead Message Queue Properties

Property	*Value*	*Editable*
Name	`SonicMQ.deadMessage`	No
Global	`false`	No
Exclusive	`false`	Yes
Save Threshold	1,536 K	Yes
Maximum Queue Size	16,384 K	Yes

The settings for save threshold and maximum queue size are highly specific to an application. Therefore, you should change these from their default settings to values appropriate to your application.

The administrator can modify all the parameters of the `SonicMQ.deadMessage` queue, except the `Name` and `Global` setting, using the Management Console. See the "Configuring Queues" chapter in the *Progress SonicMQ Configuration and Management Guide* for information about modifying the DMQ parameters.

The administrator can also modify Access Control for the DMQ through the parameter security settings. See the "Configuring Queues" chapter in the *Progress SonicMQ Configuration and Management Guide* for information about administrative modifications to Access Control for the DMQ.

JMS_SonicMQ Message Properties Used for DMQ

The message properties associated with messages declared undeliverable and possibly moving to the DMQ are the following:

- `JMS_SonicMQ_preserveUndelivered`

  Set this `boolean` property to `true` for every message that should be transferred to the `SonicMQ.deadMessage` queue when undeliverable. (Ignored for `DISCARDABLE` topic messages.) See "Setting the Message Property to Preserve If Undelivered" on page 262.

Note If a routing user does not have permissions to write to the DMQ, messages arriving from this routing node are dropped regardless of their `JMS_SonicMQ_preserveUndelivered` property (the messages do not go to the DMQ).

- `JMS_SonicMQ_notifyUndelivered`

  Set this `boolean` property to `true` for every message that should raise an administration notification when noted as being undeliverable. (Ignored for `DISCARDABLE` topic messages.)
- `JMS_SonicMQ_undeliveredReasonCode`

  Read this `int` property to determine why SonicMQ declared this message as undeliverable. The broker sets this property when messages are moved to a dead message queue.
- `JMS_SonicMQ_undeliveredTimestamp`

  Read this `long` property to determine when SonicMQ declared this message undeliverable. The broker sets this property when messages are moved to a dead message queue.

These property names are available as standard constants in `Sonic.Jms.Ext.Constants`. Table 24 provides the values for these constants.

Table 24. SonicMQ Properties

SonicMQ Constants	*String Value*
`NOTIFY_UNDELIVERED`	"`JMS_SonicMQ_notifyUndelivered`"
`PRESERVE_UNDELIVERED`	"`JMS_SonicMQ_preserveUndelivered`"
`UNDELIVERED_REASON_CODE`	"`JMS_SonicMQ_undeliveredReasonCode`"
`UNDELIVERED_TIMESTAMP`	"`JMS_SonicMQ_undeliveredTimestamp`"

Setting the Message Property to Preserve If Undelivered

To save undeliverable messages in the DMQ, a sender must set the message property `JMS_SonicMQ_preserveUndelivered` to `true`, as follows:

```
// Static setup
private const System.String Q_NAME = <Various>

// Set the msg to be preserved in the Dead Message Queue.
msg.setBooleanProperty("JMS_SonicMQ_preserveUndelivered", true);

// Create a Queue and send the message to this queue.
Sonic.Jms.Queue theQueue = session.createQueue(Q_NAME);
Sonic.Jms.MessageProducer sender = session.createProducer(null);
sender.send (theQueue, msg);
```

Handling Undelivered Messages

The following sequence of events describes how SonicMQ handles undeliverable messages:

1. A condition occurs where the broker determines the message is not deliverable.

   See "Undelivered Message Reason Codes" on page 267 for possible causes.

2. The message is passed to a special processing object in the SonicMQ broker.

   That object examines the message header.

3. The special processing object determines whether to preserve the message in the DMQ (unless the message is a `DISCARDALBE` topic message). The message is checked for the boolean property:

   ```
   JMS_SonicMQ_preserveUndelivered
   ```

   If this property is `true`, then the message is transferred to the `SonicMQ.deadMessage` queue with the following properties:

   ```
   JMS_SonicMQ_undeliveredReasonCode = reason_code [int]
   JMS_SonicMQ_undeliveredTimestamp = GMT_timestamp [long]
   ```

   See "Undelivered Message Reason Codes" on page 267 for a description of *reason_code*.

4. The special processing object determines whether to send a notification that the message was sent to the DMQ or that the message is a queue message that expired (expired topic messages are not sent to the DMQ).

   The message is checked for the boolean property:

   ```
   JMS_SonicMQ_notifyUndelivered
   ```

   If this property is `true`, an administration notification is sent with the following information:

   - Reason code
   - MessageID (of the original message)
   - Destination (of the original message)
   - Timestamp (of when the message underwent dead message handling)
   - Name of the broker (where message originated)
   - Preserved boolean (`true`, if the message was saved to the DMQ)

Sample Scenarios in Handling Dead Messages

The following sections describe typical scenarios in handling dead messages:

- “Preserving Expired Messages and Throwing an Admin Notice”
- “Using High Priority and Throwing an Admin Notice” on page 265

Preserving Expired Messages and Throwing an Admin Notice

Typically, important messages are sent `PERSISTENT` and are flagged both to be preserved on expiration and to throw an administration notification. Code Sample 9 shows how this might be done.

Code Sample 9. Preserving Expired Messages

```
// Create a TextMessage for the payload. Make sure the message
// is delivered within 2 hours (7,200,000 milliseconds).
// If expires, send a notification and save the message.
Sonic.Jms.TextMessage msg = session.createTextMessage();
msg.setText(“This is a test of notification and DMQ”);

//
// Set 'undelivered' behavior. Optionally, we could have used the
// property names defined as static const Strings in
// Sonic.Jms.Ext.Constants.
msg.setBooleanProperty("JMS_SonicMQ_preserveUndelivered", true);
msg.setBooleanProperty("JMS_SonicMQ_notifyUndelivered", true);


// Send the message with PERSISTENT, TimeToLive values.
qsender.send(msg,
             Sonic.Jms.DeliveryMode.PERSISTENT,
             Sonic.Jms.DefaultMessageProperties.DEFAULT_PRIORITY,
             7200000);
```

Using High Priority and Throwing an Admin Notice

Code Sample 10 shows how a small message can be sent using high priority, with the expectation that the message will be delivered in ten minutes. In this example, only notification events are generated.

Code Sample 10. Using High Priority

```
// Create a TextMessage for the payload. Make sure the message
// is delivered within 10 minutes (600,000 milliseconds).
// If expires, send a notification.
Sonic.Jms.TextMessage msg = session.createTextMessage();
msg.setText(“Test of undelivered events”);


// Set 'undelivered' behavior. Optionally, we could have used the
// property names defined as static const Strings in
// Sonic.Jms.Ext.Constants.
msg.setBooleanProperty("JMS_SonicMQ_notifyUndelivered", true);


// Send the message for fast delivery, or not at all.
qsender.send(msg,
             Sonic.Jms.DeliveryMode.NON_PERSISTENT,
             8,        // Expedite at a high priority
             600000);  // 10 minutes
```

What To Do When the Dead Message Queue Fills Up

When a message causes the DMQ to exceed its maximum queue size, the broker enqueues the message and then shuts down, ensuring that no messages are lost. See the "Configuring Queues" chapter in the *Progress SonicMQ Configuration and Management Guide*.

Undelivered Messages Due to Expired TTL

The reason code for a message that is undelivered due to an expired time to live (TTL) is: **`UNDELIVERED_TTL_EXPIRED`**

Note Reason codes are defined as `public static const int` in `Sonic.Jms.Ext.Constants` class.

In this case, the SonicMQ broker determines that a message expired. This failure type applies only to queue messages.

When sending messages, you can optionally set the parameter **time to live** (TTL). This TTL is converted to an expiration time and stored in the message header (in GMT).

When a SonicMQ broker tries to deliver a message, it notes the expiration time (based on the GMT as calculated from the broker's system clock) and might not deliver the message due to expiration.

Checks for expiration are done only periodically within a broker (to avoid extra overhead). Messages are always guaranteed not to be delivered if they have expired. However, the actual time they are moved to the dead message queue might be significantly later than the expiration date in the header. You can enable queue cleanup and set the cleanup interval from the Sonic Management Console in the **Edit Queues Properties** dialog box, as shown in Figure 26. See the "Configuring Queues" chapter in the *Progress*

SonicMQ Configuration and Management Guide for information about using the Sonic Management Console to set the queue cleanup parameters.

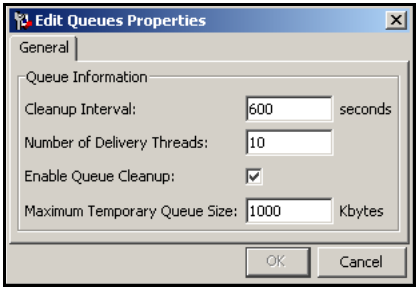


Figure 26. Queue Cleanup Interval Setting

Other cases where messages might be sent to the DMQ occur in scenarios involving dynamic routing, remote publishing, and global subscribing. See the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide*.

Undelivered Message Reason Codes

Undelivered messages can result from routing of queue and topic messages, and from HTTP direct routing extensions. The reason codes generated for these types of undelivered messages are described in this section. These reason names, `Strings` in `Sonic.Jms.Ext.Constants`, are descriptions that SonicMQ associates with undelivered messages.

Table 25 lists the reason codes for undelivered messages that can occur for both topic and queue messages, with or without dynamic routing or remote nodes.

Table 25. Reason Codes for Undelivered Messages

Reason	*Value*	*Reason Marked as Undeliverable*
`UNDELIVERED_TTL_EXPIRED`	**`1`**	The current system time on the broker (as GMT) exceeds the message's expiration time (as GMT).
`UNDELIVERED_MESSAGE_TOO_LARGE_FOR_QUEUE`	**`9`**	Message is larger than the size of the queue.

Table 26 contains a reason code that can occur only for undelivered queue messages under dynamic routing. See the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* for some examples of dynamic routing of queue messages.

Table 26. Reason Code for Undelivered Queue Routing Messages

Reason	*Value*	*Reason Marked as Undeliverable*
UNDELIVERED_ROUTING_INVALID_DESTINATION	**4**	Message received by a broker from a remote routing node has a message destination that does not exist as a global queue in the current routing node. Applies to dynamic routing of queue messages only.

Table 27 lists the reason codes that can occur for undelivered messages under dynamic routing and, in some cases as indicated, in remote publishing or subscribing. See the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* for some examples of dynamic routing of queue messages and remote publishing and subscribing.

Table 27. Reason Codes for Undelivered Routing Messages (All Domains)

Reason	*Value*	*Reason Marked as Undeliverable*
UNDELIVERED_ROUTING_INVALID_NODE	**3**	The target routing node in the destination cannot be found in the broker's list of routing connections.
UNDELIVERED_ROUTING_NOT_ENABLED	**2**	The target routing node in the destination cannot be found in the broker's list of routing connections.
UNDELIVERED_ROUTING_TIMEOUT	**5**	Message received by a broker cannot establish a remote connection to the destination routing node after trying for the specified period of time.
UNDELIVERED_ROUTING_INDOUBT	**6**	Message is unacknowledged between brokers, leaving the message in-doubt. The brokers try to re-establish the connection and resolve the situation.
UNDELIVERED_ROUTING_CONNECTION_AUTHENTICATION_FAILURE	**7**	Routing connection username and password are not authorized at a routing node while connecting to the remote broker.
UNDELIVERED_ROUTING_CONNECTION_AUTHORIZATION_FAILURE	**8**	Routing connection username does not have appropriate permissions to connect to the specified routing node (Route ACL).

Table 28 lists the reason codes that occur only for undelivered topic messages. See the "Multiple Nodes and Dynamic Routing" chapter in the *Progress SonicMQ Deployment Guide* for some examples of remote publishing of and remote subscribing to topic messages.

Table 28. Reason Codes for Undelivered Topic Routing Messages

Reason	*Value*	*Reason Marked as Undeliverable*
UNDELIVERED_ROUTING_TOPIC_MESSAGES_NOT_SUPPORTED	**18**	Message cannot be delivered to the destination because the remote node does not support remote topic messages.
UNDELIVERED_ROUTING_SUBSCRIPTION_AUTHORIZATION_FAILURE	**19**	Subscription request cannot be delivered to the destination because the remote node denies subscribe permission to the routing user.
UNDELIVERED_ROUTING_REMOTE_SUBSCRIPTION_DELETED	**20**	Message was marked undelivered because the remote subscription was deleted or expired.
UNDELIVERED_ROUTING_REMOTE_SUBSCRIPTIONS_NOT_SUPPORTED	**21**	Subscription request cannot be delivered to the destination because the remote node does not support remote subscriptions.

Table 29 lists the reason codes that can occur for undelivered HTTP Direct routing messages. See the “HTTP Direct Acceptors and Routings” chapter in the *Progress SonicMQ Deployment Guide* for information about HTTP Direct routing and examples of how to implement it in your deployments.

Table 29. Reason Codes for Undelivered HTTP Direct Routing Messages

Reason	*Value*	*Reason Marked as Undeliverable*
UNDELIVERED_HTTP_GENERAL_ERROR	**10**	Message intended for dynamic routing over HTTP was marked undeliverable for unknown reasons, or for reasons not covered by the other DMQ codes.
UNDELIVERED_HTTP_BAD_REQUEST	**12**	Message intended for dynamic routing over HTTP was rejected by the destination server because the format of the HTTP request was not valid (for example, missing a property).
UNDELIVERED_HTTP_AUTHENTICATION_FAILURE	**13**	Message intended for dynamic routing over HTTP was rejected by the destination server because the supplied username/password or certificate was invalid.
UNDELIVERED_HTTP_FILE_NOT_FOUND	**14**	Message intended for dynamic routing over HTTP was marked undelivered.
UNDELIVERED_HTTP_REQUEST_TOO_LARGE	**15**	Message intended for dynamic routing over HTTP was not sent because the HTTP request was too large.
UNDELIVERED_HTTP_INTERNAL_ERROR	**16**	Message intended for dynamic routing over HTTP was not sent because the destination service was unable to process the request.

Normally, HTTP Direct routing extensions are used when a SonicMQ broker is sending to a non-Sonic Web server. However, there is nothing to stop you from using HTTP Direct to talk to another broker that has inbound HTTP Direct acceptors. In this case, the additional errors listed in Table 30 might occur.

Table 30. Additional Reason Codes for Undelivered HTTP Direct Routing Messages

Reason	*Value*	*Reason Marked as Undeliverable*
`UNDELIVERED_HTTP_PROTOCOL_NOT_SUPPORTED`	`17`	Message intended for dynamic routing over HTTP was marked undelivered because the request was sent to an unregistered URL (there is no protocol handler listening for requests on that URL).
`UNDELIVERED_HTTP_HOST_UNREACHABLE`	`11`	Message intended for dynamic routing over HTTP was marked undelivered for one of the following reasons: ● A connection cannot be made to the HTTP destination ● The request timed out

Chapter 11 Hierarchical Name Spaces

Hierarchical name spaces allow you to create a hierarchy of contents by delimiting nodes when you name a topic. You can use this feature to name and manage topics. This chapter contains the following sections:

About Hierarchical Name Spaces

Hierarchical name spaces are a topic-grouping mechanism available with SonicMQ. SonicMQ extends topic management in a way that adds virtually no overhead when publishing, yet provides faster access, easier filtering, and flexible subscriptions. By delimiting nodes when naming a topic, a hierarchy of contents is created at the broker. This chapter describes how and when to use hierarchical name spaces.

Naming conventions become cumbersome to work with when long strings are passed around as identifiers. SonicMQ offers the ability to use a naming and directory service with the naming and management of topics. As a result, topics are easier to specify and control for clients and are correspondingly faster to manage and control by the broker.

While a topic hierarchy can be flat (linear), it typically builds from one or more root topics, adding other topics in levels of parent-child relationships to create a hierarchical naming structure.

The SonicMQ administrator can set and monitor security with the same template character devices to assure that the scope of message permissions is appropriate for each user individually and as a member of one or more groups. See the *Progress SonicMQ Deployment Guide* to learn how security can control access to topic name spaces.

In most messaging systems, there is a one-level structure, as shown in Figure 27.

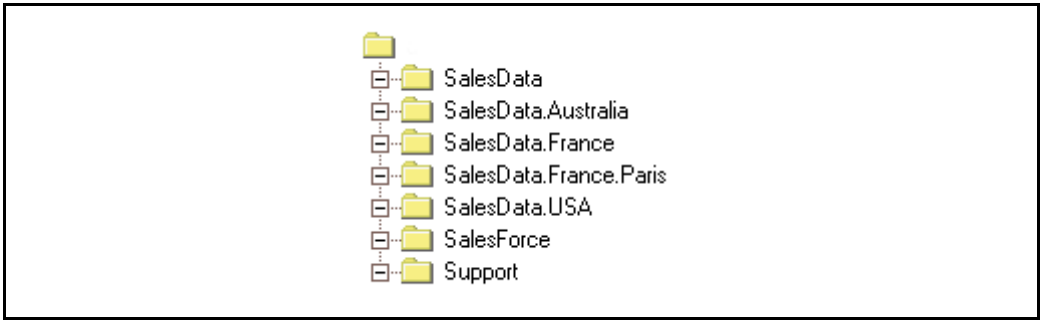


Figure 27. Topic Structure Without Hierarchies

Without hierarchies, many topics are stacked onto one level. When many topics are used, it becomes increasingly difficult to maintain access to the naming structure and to denote topic relationships.

Hierarchical name spaces in SonicMQ use a parent-child subordinated folder structure, as shown in Figure 28.

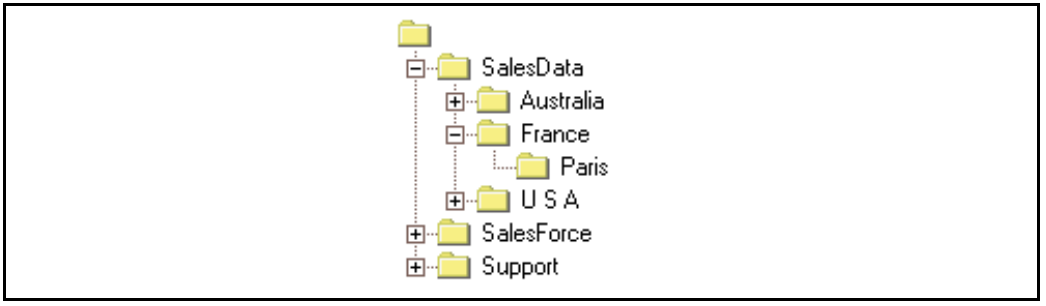


Figure 28. Topic Structure with Hierarchies

With hierarchies, a topic named **`SalesData.France.Paris`** denotes a content node in a hierarchical structure that can participate in selection mechanisms that refer to its depth in the structure (third-level), the name of the node itself (**Paris**), and its memberships (**Paris** is a member of **France** and a member of **SalesData**, among others).

Meaningful names in a topic hierarchy offer many other advantages for message retrieval and security authorization, as discussed later in this chapter.

Publishing Messages to Topics

Structuring useful topic hierarchies optimizes the management of the hierarchy for the broker and its accessibility by subscribers.

Publishing a message to a topic encourages the use of hierarchy delimiters and deprecates the use of a few special characters and topic names.

Hierarchical name spaces use the same notation as fully qualified packages and classes—period-delimited strings. Security controls whether or not an authenticated user has permission to publish to a topic content node.

See the *Progress SonicMQ Deployment Guide* to learn how security can control publication to topic content nodes.

Reserved Characters When Publishing

Some characters and strings are reserved for special use:

- Delimit the hierarchical nodes with **.** (period). For example, the `Chat` sample uses the topic name, **`jms.samples.chat`**.
- Do not use * (asterisk), $ (dollar sign), or # (pound) in topic names.
- Reserve **`$SYS`** and **`$ISYS`** for administrative topics.

Topic Structure, Syntax, and Semantics

There are few constraints on a topic hierarchy. SonicMQ supports:

- Unlimited number of topics at any content node
- Unlimited depth of the hierarchy (period-delimited strings)
- Unlimited number of topic hierarchies
- Long name for any topic node and any topic
- Long name for the complete string that defines a specific node

Compact, balanced structures always outperform bulky hierarchical structures. There are, however, some naming constraints:

- The name must be one or more characters in length with neither leading nor trailing blank space. Embedded spaces are acceptable.
- The topic hierarchies rooted at **$SYS** and **$ISYS** are reserved for the broker's system messages.

Note For more information on **$SYS** and **$ISYS**, see the *Progress SonicMQ Configuration and Management Guide*.

Topic Syntax and Semantics

The following naming conventions apply to topic naming:

- **Case sensitive** — Topic names are case sensitive. For example, SonicMQ recognizes **ACCOUNTS** and **Accounts** as two different topic names.
- **Spaces in names** — Topic names can include the space character. For example, **accounts payable.** Spaces are treated just like any other character in the topic name.
- **Empty string** — A topic level can be an empty string. For example, **a..c** is a three-level topic name whose middle level is empty. The root node is not a content node, so just an empty string (**" "**) is not a valid topic level for publication.

Note The value `null` indicates an absence of content, or a zero-length string. The Unicode `null` character (`\x0000`) is not a `null` in this convention.

Broker Management of Topic Hierarchies

Topic hierarchies empower the broker in two significant ways:

- Selection and filtering of topics is, for most purposes, already accomplished. Access to multiple topics is indexed for much faster retrieval than flat naming systems.
- Security otherwise set for each topic individually can be established for a content node and, optionally, its subordinate nodes.

Subscribing to Nodes in the Topic Hierarchy

Subscriptions are created in the standard way with the **`Topic`** and the **`TopicSubscriber`** methods. As shown in Figure 29, to receive messages published for **U S A Credit**, use the topic name **`Credit.U S A`**.

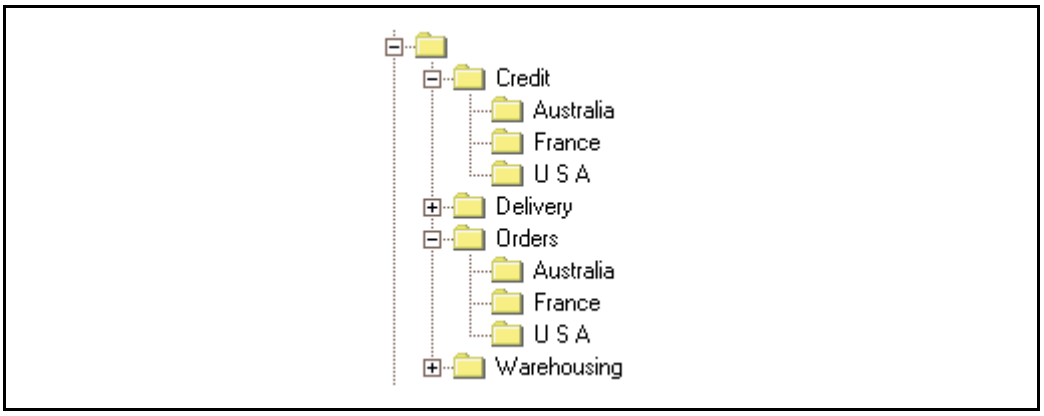


Figure 29. Subscribing to the Topic Credit.U S A

While hierarchical topics enable powerful security and accelerate the retrieval of topics by the broker, SonicMQ topic hierarchies enable unique multiple topic subscriptions, allowing you to:

- Subscribe to many topics quickly
- Subscribe to topics whose complete name is unknown
- Traverse topic structures in powerful ways

When you use topic hierarchies, message selectors—an inherently slow and recurring process—can often be eliminated.

Template Characters

Wild cards are special characters in a sample string that are interpreted when evaluating a set of strings to form a list of qualified names. In this case, however, the special characters are referred to as **template characters** because the entire string and its special characters can be stored for later evaluation by durable subscriptions and security permissions. The selection of topic names is dynamic, evaluated every time the topic is requested.

The period (.) delimiter is used together with the asterisk (*) and the pound (#)template characters when subscriptions are fulfilled. Using these characters avoids having to subscribe to multiple topics and offers benefits to managers who might need to see information or events across several areas. Client applications can only use template characters when subscribing to a set of topics or binding a set of topics to a message handler. Messages must be published on fully specified topic names.

Using template characters is somewhat different from using the usual wild cards, as discussed below.

There are two SonicMQ template characters:

- asterisk (*) — Selects all topics at this content node.
- pound (#) — Selects all topics at this content node and the subordinate hierarchy (when used in the end position) or the superior hierarchy (when used in the first position).

Using template characters allows a set of managed topics to exist in a message system in a way that lets subscribers choose broad subscription parameters that include preferred topics and avoid irrelevant topics.

There are some constraints:

- Unlike shell searches, you cannot qualify a selection, such as **`Alpha.B*.Charlie`**. You can use **`Alpha.*.Charlie`**. At a content level, a template character precludes using other template characters.
- The # symbol can only be used once. It is placed in the first node position or in the last node position. You can use **`Alpha.#`**, or **`*.*.Charlie.#`**, or **`#.Beta.Charlie`**, or just **`#`**, but not **`#.Beta.#`**. If you use only **`#`**, you receive not only messaging traffic, but also management messages sent between the domain manager and the broker.
- Character replacement, as used in shell searches with the question mark character (?), is not allowed.

SonicMQ delivers a message to more than one message handler if the message's topic matches bindings from multiple handlers.

The content levels in the topic name space consider the root level **`""`** as level **`0`**.

Using Template Characters in Symmetric Hierarchies

When hierarchical structures are strictly defined, simple templates can be used. For example, the topic hierarchy shown in Figure 30 appears to strictly assign business functions—**Credit**, **Delivery**, **Orders**, and **Warehousing**—to first-level (parent) nodes and a standard set of country names—**Australia**, **France**, **USA**—to second-level (child) nodes.

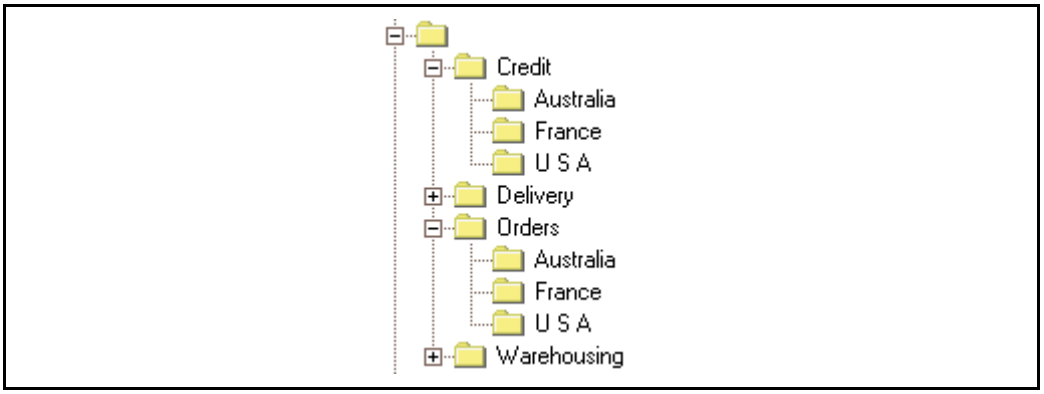


Figure 30. Symmetric Topic Structure

Template Character for All Topics at a Content Level

Using the strict topic hierarchy shown in Figure 30, a client application could subscribe to each of the three topic nodes for **`Credit`**.

By using a template character, the application can subscribe to all second-level **`Credit`** topics by subscribing to **`Credit.*`**, a subscription that delivers messages sent to these destinations:

- **`Credit.Australia`**
- **`Credit.France`**
- **`Credit.U S A`**

Template Character for a Topic at a Content Level

A subscription to the topic expression **`*.U S A`** in the hierarchy in Figure 30 selects all **`U S A`** topics at the second level of the hierarchy.
This subscription delivers messages sent to these destinations:

- **`Credit.U S A`**
- **`Orders.U S A`**

Using Template Characters in Asymmetric Topic Hierarchies

When there are several topic levels, as shown in Figure 31, subscribing to all the **U S A** topics is complicated by an inconsistent topic-naming structure.

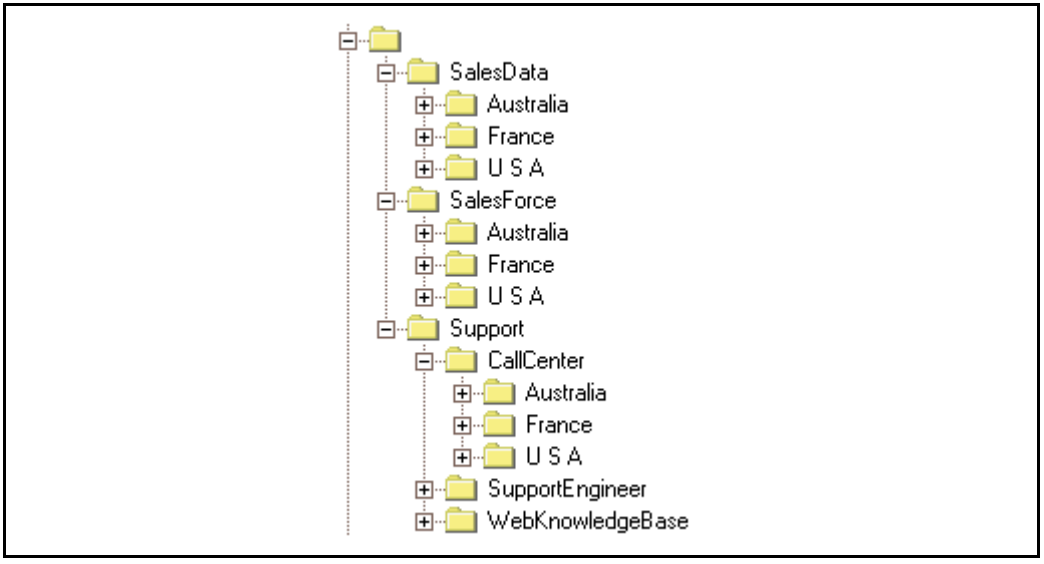


Figure 31. Asymmetric Topic Structure

In this case, the **#** template character can be used to subscribe to the **U S A** topic levels in the hierarchy regardless of intervening nodes, so that **#.U S A** subscribes to topics at these destinations:

- **SalesData.U S A**
- **SalesForce.U S A**
- **Support.CallCenter.U S A**

Without this ability, you have to subscribe to both ***.U S A** and ***.*.U S A** to create the same subscriptions.

Note When you use the **"#"** template character as the leading character in an expression, you can inadvertently reveal messages in unseen lower levels.

Template Character for Subscribing to All Topics

Subscribing to the topic name **#** receives all messages, including the reserved system topics **$SYS** and **$ISYS**.

Template Character for All Topics Under a Topic Hierarchy

When it is not known how deep the topic structure extends and all subordinate topics are of interest, appending *name*.# extends the subscriptions to all topics at or below that level—for example, **Support.#** subscribes to:

- **Support.CallCenter**
- **Support.CallCenter.Australia**
- **Support.CallCenter.France**
- **Support.CallCenter.U S A**
- **Support.SupportEngineer**
- **Support.WebKnowledgeBase**

plus any subordinate levels below those topic nodes.

The **MessageMonitor** sample displays all the messages published on the broker host by subscribing to **jms.samples.#**. The sample does not subscribe to **#**, because such a subscription includes management messages that are not relevant to the sample.

Template Character for All Topics Above a Topic Hierarchy

When the deepest node name is known but the number of intervening levels is not, using #.*name* enables any number of levels to be accessed. For example, **#.Support** subscribes to:

- **Online.Europe.Support**
- **ISV.EMEA.France.Paris.Support**
- **HQ.Support**

Multiple Template Characters in an Expression

Some template characters can be combined in a single expression—for example:

- Use only one template character at a topic level. (**Support.**.U S A** is invalid.)
- Use the pound sign only once in an expression. (**#.U S A.# is** invalid.)

Examples of multiple template characters in an expression are:

- Use **`#.U S A.*`** to subscribe to just the topics at **`U S A`** nodes however deep in the topic structure, but not messages at **`#.U S A`**.
- Use **`*.*.U S A.*`** to subscribe to just the topics at level 4 **`U S A`** nodes, but not those at **`*.*.U S A`**.

Examples of Topic Name Spaces

The hypothetical topic hierarchy shown in Figure 32 has nodes that might represent levels of responsibility in the enterprise.

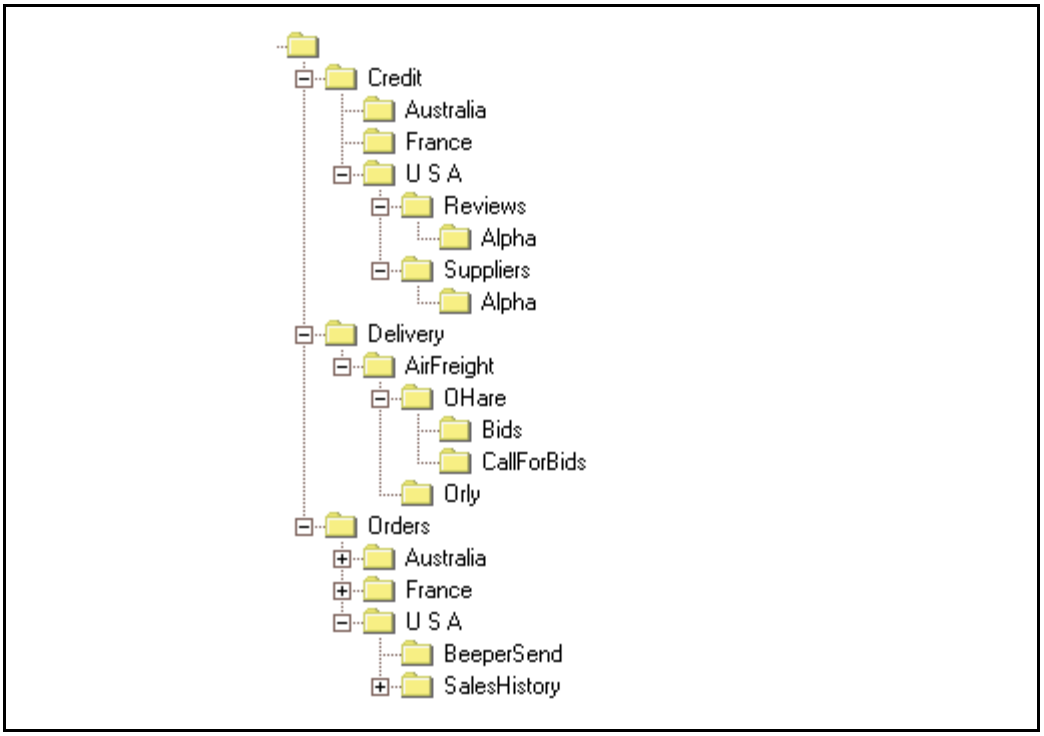


Figure 32. A Sample Hierarchy of Topics

Publishing Messages to a Hierarchical Topic

The publisher produces messages to a single fully qualified topic, such as:

```
static const System.String MESSAGE_TOPIC = "Credit.U S A.Customers";
```

Business cases where a publisher might use hierarchical topics are:

- Requests for regular credit updates about suppliers are routed to **`Credit.U S A.Suppliers`** and use `JMSReplyTo` mechanisms.
- Messages that are sent to credit agencies at secure Internet topics **`Credit.U S A.Customers`** and **`Credit.U S A.Suppliers`** should be accessible only by authorized applications.
- Credit agencies can respond to credit requests through the special topic **`Credit.U S A.Reviews`**. Use a **`Reviews`** topic to get secure responses to credit requests without synchronous blocks.
- As orders are processed through application software, any problems or delays send a message to the appropriate sales force beeper number listed in the application. The message producer uses the topic **`Orders.U S A.BeeperSend`**, attaching the beeper number as the `JMSCorrelationID` or SonicMQ-supplied message property.
- Messages are sent that outline expected shipping needs to topics like **`#.Ohare.CallForBids`**.

Subscribing to Sets of Hierarchical Topics

Subscribers to topics can also specify a fully qualified topic:

```
private static const System.String MESSAGE_TOPIC = "Credit.U S A";
```

or use template characters to subscribe to sets of topics:

```
private static const System.String MESSAGE_TOPIC = "Credit.*";
```

Business cases where a subscriber gains an advantage by using template characters to subscribe to hierarchical topics are:

- Accounting subscribes to **`Credit.U S A.Customers.Reviews`**, but the auditor subscribes to **`Credit.U S A.#`** to watch all credit activity.
- By listening to **`Credit.U S A.*.Reviews`**, the application gets only the U S A responses to all types of credit requests without synchronous blocks.
- A communications service monitors the brokers at its limited-access read-only topics, **`*.U S A.BeeperSend`** and then executes the download.

Chapter 12 COM Client to C# Client Migration

If you have developed client applications in C# using the SonicMQ COM client, you can migrate those applications to C# in the .NET client, which is more feature-rich than the COM client. This chapter describes issues to consider during migration; it contains the following sections:

Overview

Existing COM clients use the COM library exported as a C# DLL; this library is generated by running the `tlbimp.exe` on the COM library. The new .NET client is a completely managed client written in the C# language.

Namespace Changes

Existing COM clients import the `JMSCOMCLIENTLib` by using the following C# directive:

```
using JMSCOMCLIENTLib
```

To migrate successfully, these clients must replace the above directive with appropriate C# client namespace (or namespaces). For example:

```
using Sonic.Jms
```

For more information about the namespaces provided with the C# client, see "SonicMQ .NET Client Application Programming Interfaces" on page 21.

Interface Changes

To migrate successfully, existing COM clients must replace COM interface names with corresponding C# interface names.

COM Interface	*C# Interface*
Messages	
`CJMSMessage`	`Message`
`IJMSTextMessage`	`TextMessage`
`IJMSBytesMessage`	`BytesMessage`
Connection Factories	
`IJMSTopicConnectionFactory`	`TopicConnectionFactory`
`IJMSQueueConnectionFactory`	`QueueConnectionFactory`
Connections	
`IJMSTopicConnection`	`TopicConnection`
`IJMSQueueConnection`	`QueueConnection`

COM Interface	*C# Interface*
Sessions	
IJMSTopicSession	TopicSession
IJMSQueueSession	QueueSession
Message Producers	
IJMSTopicPublisher	TopicPublisher
IJMSQueueSender	QueueSender
Message Consumers	
IJMSTopicSubscriber	TopicSubscriber
IJMSQueueReceiver	QueueReceiver
Queue Browsers	
IJMSQueueBrowser	QueueBrowser
Destinations	
IJMSTopic	Topic
IJMSQueue	Queue
Message Listeners	
IJMSMessageListener	MessageListener
Exception Listeners	
IJMSExceptionListener	ExceptionListener
Exceptions	
IJMSExceptionListener	ExceptionListener

ConnectionFactory Objects

Existing COM clients create a topic or queue connection factory by creating a new `CJMSTopicConnectionFactory` object or `CJMSQueueConnectionFactory` object. For example:

- `IJMSTopicConnectionFactory tcf = new CJMSTopicConnectionFactory;`
- `IJMSQueueConnectionFactory qcf = new CJMSQueueConnectionFactory;`

To migrate successfully, existing clients must replace these objects with the following:

- `Sonic.Jms.Cf.Impl.TopicConnectionFactory`
- `Sonic.Jms.Cf.Impl.QueueConnectionFactory`.

For example:

```
TopicConnectionFactory tcf = new Sonic.Jms.Cf.Impl.TopicConnectionFactory();
```

Contrasting V6.0 COM and V7.6.2 .NET Client Samples

If you have an installation of a V6.0 or earlier Sonic COM client, you can review the changes necessary to migrate from the COM client to the C# client by comparing the source code of the C# samples that are shipped in both products.

In the V6.0 or earlier COM client, the functionally comparable samples are located at *`cclients_install_dir`*`\samples\COM\CSharp`:

- `QueuePTP\GlobalTalk`
- `TopicPubSub\DurableChat`
- `TopicPubSub\RequestReply`

In the Progress SonicMQ V7.6.2 .NET client, C# samples of the name are located at: *`net_client_install_dir`*`\samples\csharp.`

Appendix A

Comparing the SonicMQ .NET Client with the Java Client

This appendix distinguishes which SonicMQ client features are available in the SonicMQ Java Client and that are not yet available in the SonicMQ .NET Client.

Messaging Features

Some features of the SonicMQ Java Client have not been ported to other clients. If any of these features suits your architectural needs, contact your Progress Software representative to learn about the feature's scheduled implementation in the SonicMQ .NET Client.

Sonic Stream API

This API enables a stream publisher application to send a stream of bytes of indeterminate length as segments through a JMS broker topic to many stream subscribers. See the "SonicStream API" chapter in the *Progress SonicMQ Application Programming Guide*. This feature is not supported by the Progress SonicMQ .NET client.

Programmatic Limit to Redelivery Attempts from a Queue

Applications can exit a loop where a message causes application failure and rollback, only to repeat the failure when the same "poison message" is redelivered. The feature enables programmatic setting of a maximum redelivery count that causes unacknowledged messages that exceed the redelivery limit to be discarded. See the "SonicMQ Client Sessions" chapter in the *Progress SonicMQ Application Programming Guide*. This feature is not supported by the Progress SonicMQ .NET client.

Non-Persistent Replicated Delivery Mode (CAA-FastForward)

SonicMQ provides a delivery mode (`NON_PERSISTENT_REPLICATED`) that protects non-persistent messages from broker failures by replicating the messages to a standby broker. This feature is called CAA-Fast Forward (CAA-FF). CAA-FF combines the performance of non-persistent messaging with the reliability of Sonic's Continuous Availability Architecture to provide an unparalleled message throughput and latency for reliable delivery. See the "SonicMQ Connections" chapter in the *Progress SonicMQ Application Programming Guide*. This feature is not supported by the Progress SonicMQ .NET client.

MultiTopic Constructs for Producers and Consumers

Application code or administered destination objects that define a list of topics as a MultiTopic enable publishers to accelerate the publishing operation to many topics concurrently, even to a specified Dynamic Routing node. A new sample, `MultiTopicChat`, is included. See the "Publish and Subscribe Messaging" and "Examining the SonicMQ JMS Samples" chapters in the *Progress SonicMQ Application Programming Guide*. This feature is not supported by the Progress SonicMQ .NET client.

Socket Connect Timeout

When using a JVM that supports socket connect timeout, you can set a timeout to be used when establishing a socket connection to a broker. See the "Introduction to Configuration" chapter in the *Progress SonicMQ Configuration and Management Guide*.

This feature is also included as a parameter of the JMS Administered Objects Connection Factory. See the "SonicMQ Connections" chapter in the *Progress SonicMQ Application Programming Guide*.

This feature is not supported by the Progress SonicMQ .NET client.

Custom Destinations for Undelivered Messages

While every SonicMQ broker provides a Dead Message Queue (DMQ), you can now override the default destination for preserving undelivered messages programmatically by defining an undelivered destination. When delivery of the message to the user-specified DMQ destination fails, the message is placed in the default DMQ of the appropriate broker. See the "Guaranteeing Messages" chapter in the *Progress SonicMQ Application Programming Guide*. This feature is not supported by the Progress SonicMQ .NET client.

Shared Durable Subscriptions

Multiple subscribers can share a durable subscription in a common message store so that disconnected durable subscribers do not accrue messages that connected durable subscribers in the group are prepared to receive. See the "Publish and Subscribe Messaging" chapter in the *Progress SonicMQ Application Programming Guide* and the "Managing SonicMQ Broker Activities" chapter in the *Progress SonicMQ Configuration and Management Guide*. This feature is not supported by the Progress SonicMQ .NET client.

Asynchronous Message Delivery

A message producer can increase performance for non-transacted sessions by setting asynchronous delivery mode on the producer's connection. See the "Asynchronous Message Delivery" section of the "SonicMQ Connections" chapter of the *Progress SonicMQ Application Programming Guide* for details. This feature is not supported by the Progress SonicMQ .NET client.

Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W

X